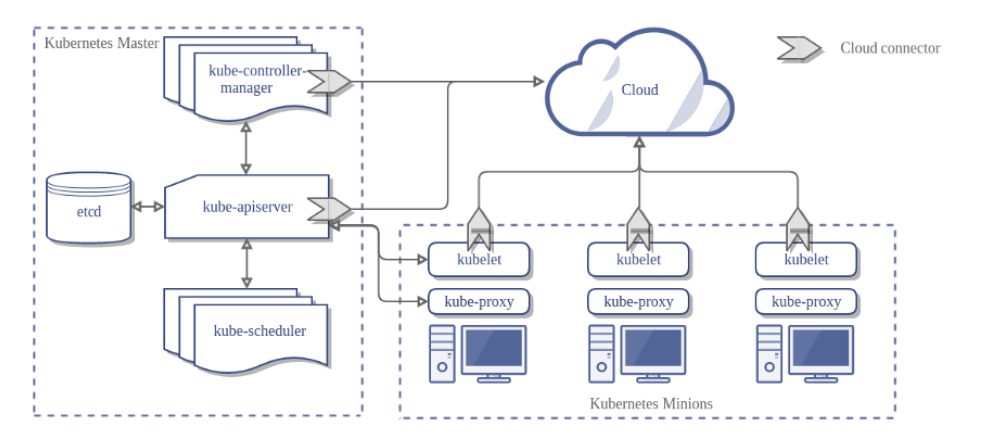
**Kubernetes**

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. Google open-sourced the Kubernetes project in 2014. Kubernetes combines [over 15 years of Google's experience](https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/) running production workloads at scale with best-of-breed ideas and practices from the community.

Let's take a look at why Kubernetes is so useful by going back in time.



**Traditional deployment era:** Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

**Virtualized deployment era:** As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application.

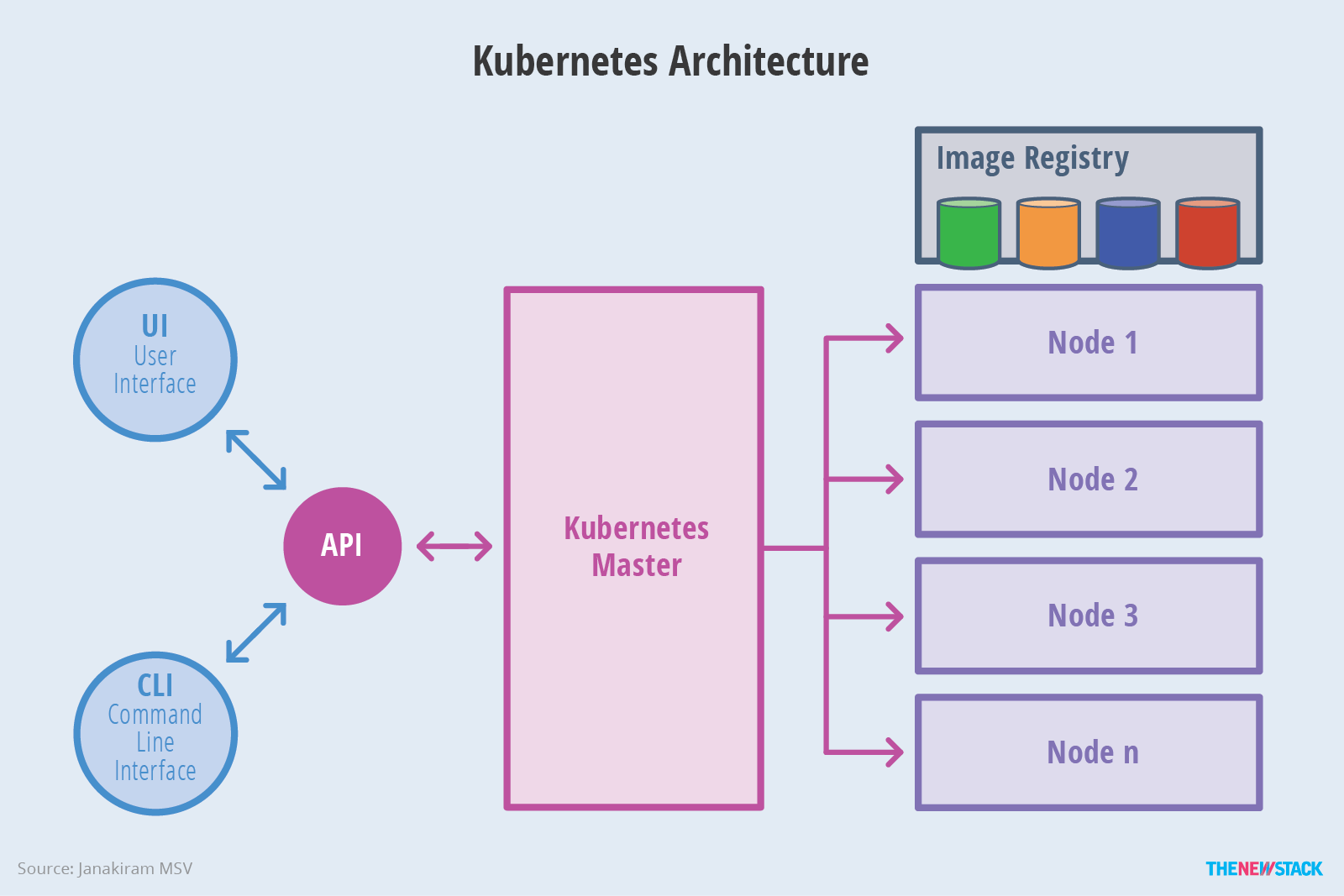
Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines.

Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

**Container deployment era:** Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

* Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
* Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
* Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
* Observability not only surfaces OS-level information and metrics, but also application health and other signals.
* Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.
* Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
* Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
* Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
* Resource isolation: predictable application performance.
* Resource utilization: high efficiency and density.

****

# Kubernetes Components

A Kubernetes cluster consists of the components that represent the control plane and a set of machines called nodes.

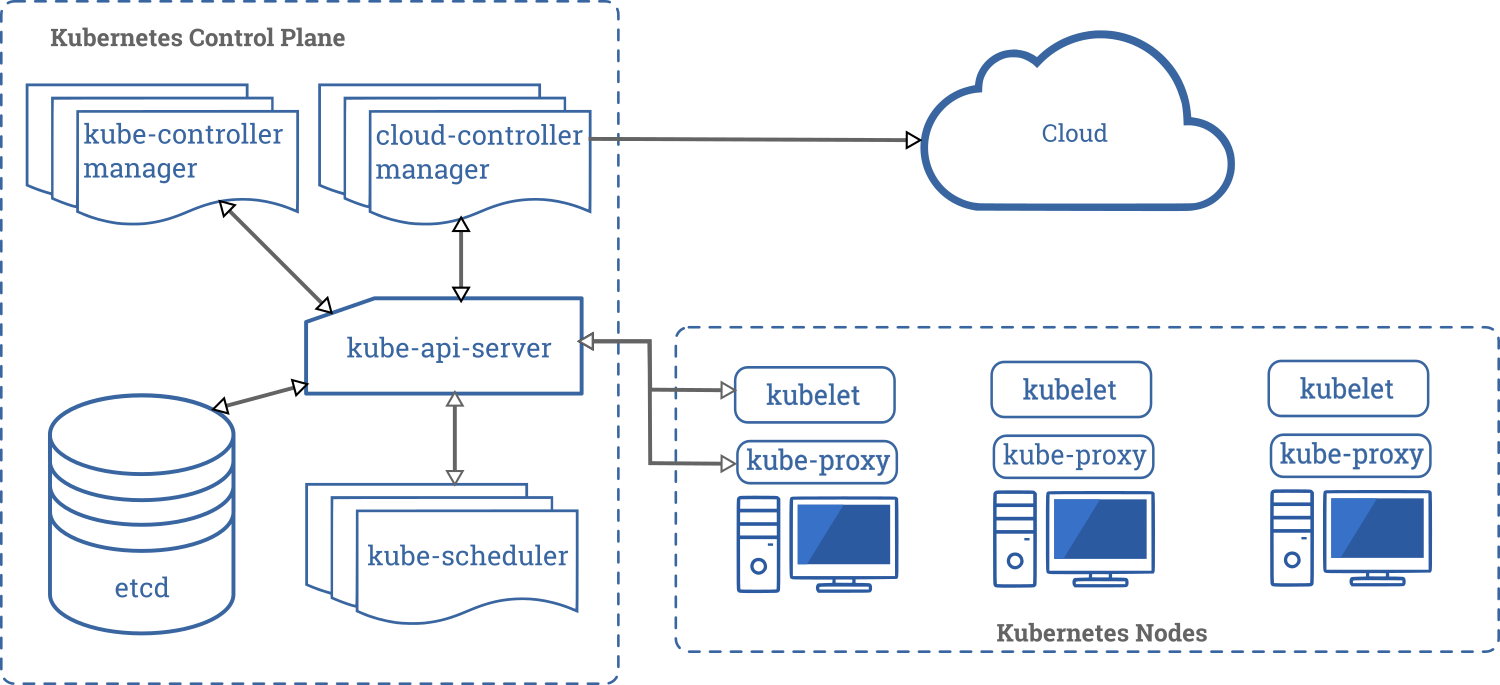
When you deploy Kubernetes, you get a cluster.

A Kubernetes cluster consists of a set of worker machines, called [nodes](https://kubernetes.io/docs/concepts/architecture/nodes/), that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) that are the components of the application workload. The [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

This document outlines the various components you need to have a complete and working Kubernetes cluster.

Here's the diagram of a Kubernetes cluster with all the components tied together.



## **Control Plane Components**

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new [pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) when a deployment's replicas field is unsatisfied).

Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See [Building High-Availability Clusters](https://kubernetes.io/docs/admin/high-availability/) for an example multi-master-VM setup.

### **kube-apiserver**

The API server is a component of the Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

The main implementation of a Kubernetes API server is [kube-apiserver](https://kubernetes.io/docs/reference/generated/kube-apiserver/). kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

### **etcd**

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a [back up](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/" \l "backing-up-an-etcd-cluster) plan for those data.

You can find in-depth information about etcd in the official [documentation](https://etcd.io/docs/).

### **kube-scheduler**

Control plane component that watches for newly created [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) with no assigned [node](https://kubernetes.io/docs/concepts/architecture/nodes/), and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

### **kube-controller-manager**

Control Plane component that runs [controller](https://kubernetes.io/docs/concepts/architecture/controller/) processes.

Logically, each [controller](https://kubernetes.io/docs/concepts/architecture/controller/) is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

* Node controller: Responsible for noticing and responding when nodes go down.
* Replication controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
* Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
* Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.

### **cloud-controller-manager**

A Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that just interact with your cluster.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

* Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
* Route controller: For setting up routes in the underlying cloud infrastructure
* Service controller: For creating, updating and deleting cloud provider load balancers

## **Node Components**

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

### **kubelet**

An agent that runs on each [node](https://kubernetes.io/docs/concepts/architecture/nodes/) in the cluster. It makes sure that [containers](https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#why-containers) are running in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/).

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn’t manage containers which were not created by Kubernetes.

### **kube-proxy**

kube-proxy is a network proxy that runs on each [node](https://kubernetes.io/docs/concepts/architecture/nodes/) in your cluster, implementing part of the Kubernetes [Service](https://kubernetes.io/docs/concepts/services-networking/service/) concept.

[kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/) maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

### **Container runtime**

The container runtime is the software that is responsible for running containers.

Kubernetes supports several container runtimes: [Docker](https://docs.docker.com/engine/), [containerd](https://containerd.io/docs/" \t "_blank), [CRI-O](https://cri-o.io/#what-is-cri-o), and any implementation of the [Kubernetes CRI (Container Runtime Interface)](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md).

## **Addons**

Addons use Kubernetes resources (Daemon Set, [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/), etc) to implement cluster features. Because these are providing cluster-level features, name spaced resources for addons belong within the kube-system namespace.

Selected addons are described below; for an extended list of available git addons, please see [Addons](https://kubernetes.io/docs/concepts/cluster-administration/addons/).

### **DNS**

While the other addons are not strictly required, all Kubernetes clusters should have [cluster DNS](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/), as many examples rely on it.

Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

Containers started by Kubernetes automatically include this DNS server in their DNS searches.

### **Web UI (Dashboard)**

[Dashboard](https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/) is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

### **Container Resource Monitoring**

[Container Resource Monitoring](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/) records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

### **Cluster-level Logging**

A [cluster-level logging](https://kubernetes.io/docs/concepts/cluster-administration/logging/) mechanism is responsible for saving container logs to a central log store with search/browsing interface.

# Deploy Load Balancing on Kubernetes Cluster on Ubuntu 18.04 LTS

## Requirements

* Two servers with Ubuntu 18.04 installed.
* Minimum 2 GB of RAM installed on each server.
* A root password is configured on both servers.

## Getting Started

First, you will need to update both servers with the latest stable version. You can update them by running the following command:

**apt-get update -y  
apt-get upgrade -y**

Once both servers are updated, restart them to apply all the changes.

By default, Kubernetes does not support swap memory and will not work if swap is active. So you will need to disable swap memory on both servers.

To disable swap memory temporary run the following command:

swapoff -a

To disable swap memory permanently open /etc/fstab file:

**nano /etc/fstab**

Comment out the last line:

# /etc/fstab: static file system information.  
#  
# use 'blkid' to print the universally unique identifier for a  
# device; this may be used with uuid= as a more robust way to name devices  
# that works even if disks are added and removed. see fstab(5).  
#  
# &lt;filesystem&gt; &lt;mountpoint&gt; &lt;type&gt; &lt;options&gt; &lt;dump&gt; &lt;pass&gt;  
# / was on /dev/sda1 during installation  
# swap was on /dev/sda4 during installation

#UUID=65se21r-1d3t-3263-2198-e564c275e156 none swap sw 0 0

Save and close the file. Then, run the following command to apply the configuration changes:

**mount -a**

Next, you will need to setup hostname resolution on both servers. So, each server can communicate with each other using the hostname.

To do so, open /etc/hosts file using your preferred editor:

**nano /etc/hosts**

Add the following lines:

Public ip master

Public ip slave1

Public ip slave2

Save and close the file, when you are finished. Then, proceed to the next step.

## Install Docker And Kubernetes

Next, you will need to install Docker and Kubernetes tool kubelet, kubeadm, and kubectl on both servers.

First, install required packages and add GPG key with the following command:

**apt-get install apt-transport-https ca-certificates curl software-properties-common -y  
 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -**

Next, add Docker CE repository on both servers by running the following command:

**add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"**

Next, update the repository and install Docker CE with the following command:

**apt-get update -y  
apt-get install docker-ce -y**

Once the installation is completed, check the status of Docker CE with the following command:

**systemctl status docker**

You should see the following output:

? docker.service - Docker Application Container Engine

Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)

Active: active (running) since Fri 2019-07-19 07:05:50 UTC; 1h 24min ago

Docs: https://docs.docker.com

Main PID: 3619 (dockerd)

Tasks: 8

CGroup: /system.slice/docker.service

??3619 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Jul 19 07:05:48 master dockerd[3619]: time="2019-07-19T07:05:48.574491681Z" level=warning msg="Your kernel does not support swap memory limit"

Jul 19 07:05:48 master dockerd[3619]: time="2019-07-19T07:05:48.575196691Z" level=warning msg="Your kernel does not support cgroup rt period"

Jul 19 07:05:48 master dockerd[3619]: time="2019-07-19T07:05:48.575733336Z" level=warning msg="Your kernel does not support cgroup rt runtime"

Jul 19 07:05:48 master dockerd[3619]: time="2019-07-19T07:05:48.582517104Z" level=info msg="Loading containers: start."

Jul 19 07:05:49 master dockerd[3619]: time="2019-07-19T07:05:49.391255541Z" level=info msg="Default bridge (docker0) is assigned with an IP add

Jul 19 07:05:49 master dockerd[3619]: time="2019-07-19T07:05:49.681478822Z" level=info msg="Loading containers: done."

Jul 19 07:05:50 master dockerd[3619]: time="2019-07-19T07:05:50.003776717Z" level=info msg="Docker daemon" commit=0dd43dd graphdriver(s)=overla

Jul 19 07:05:50 master dockerd[3619]: time="2019-07-19T07:05:50.009892901Z" level=info msg="Daemon has completed initialization"

Jul 19 07:05:50 master systemd[1]: Started Docker Application Container Engine.

Jul 19 07:05:50 master dockerd[3619]: time="2019-07-19T07:05:50.279284258Z" level=info msg="API listen on /var/run/docker.sock"

Kubernetes packages are not available in the Ubuntu 18.04 default repository. So, you will need to add the Kubernetes repository on both servers.

You can add it with the following commands:

**curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -  
 echo 'deb http://apt.kubernetes.io/ kubernetes-xenial main' | tee /etc/apt/sources.list.d/kubernetes.list**

Next, update the repository and install Kubernetes packages with the following command:

**apt-get install kubeletkubeadmkubectl -y**

Once all the packages are installed, you can proceed to configure Master server.

## Configure Kubernetes Master Server

First, you will need to initialize your cluster with its private IP address on the Master server:

You can do it with the kubeadm command:

**kubeadminit --pod-network-cidr=192.168.0.0/16 --apiserver-advertise-address=192.168.0.103**

Once the Cluster initialized successfully, you should see the following output:

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

**mkdir -p $HOME/.kube**

**sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config**

**sudochown $(id -u):$(id -g) $HOME/.kube/config**

You should now deploy a pod network to the cluster.

Run "**kubectl apply -f [podnetwork].yaml**" with one of the options listed at:

https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

**kubeadm join 192.168.0.103:6443 --token zsyq2w.c676bxzjul3upd7u \**

**--discovery-token-ca-cert-hash sha256:a720ae35d472162177f6ee39de758a5de40043f53e4a3e00aefd6f9832f3436c**

Next, you will need to configure the kubectl tool on your Master server. You can do it with the following command:

**Mkdir -p $HOME/.kube  
 cp -i /etc/kubernetes/admin.conf $HOME/.kube/config  
chown $(id -u):$(id -g) $HOME/.kube/config**

Next, you will need to deploy a Container Networking Interface (CNI) on your server. Because, the cluster does not have a CNI.

You can deploy the CNI to your cluster with the following command:

**kubectl apply -f https://docs.projectcalico.org/v2.6/getting-started/kubernetes/installation/hosted/kubeadm/1.6/calico.yaml**

You should see the following output:

configmap/calico-config created

daemonset.extensions/calico-etcd created

service/calico-etcd created

daemonset.extensions/calico-node created

deployment.extensions/calico-kube-controllers created

deployment.extensions/calico-policy-controller created

clusterrolebinding.rbac.authorization.k8s.io/calico-cni-plugin created

clusterrole.rbac.authorization.k8s.io/calico-cni-plugin created

serviceaccount/calico-cni-plugin created

clusterrolebinding.rbac.authorization.k8s.io/calico-kube-controllers created

clusterrole.rbac.authorization.k8s.io/calico-kube-controllers created

serviceaccount/calico-kube-controllers created

You can now check your namespaces by running the following command:

**kubectl get namespaces**

If everything goes fine, you should see the following output:

NAME STATUS AGE

default Active 4h

kube-public Active 4h

kube-system Active 4h

Next, verify whether the master node is now running properly with the following command:

**kubectl get nodes**

You should see the following output:

name status roles age version

master Ready master 12m v1.15.3

## Add Slave to the Kubernetes Cluster

Next, log in to your slave server and run the following command to add the slave to the Kubernetes cluster:

**kubeadm join 192.168.0.103:6443 --token zsyq2w.c676bxzjul3upd7u --discovery-token-ca-cert-hash sha256:a720ae35d472162177f6ee39de758a5de40043f53e4a3e00aefd6f9832f3436c**

Next, go to the master server and check whether the slave is added to your Kubernetes cluster with the following command:

**kubectl get nodes**

You should see the following output:

name status roles age version

master ready master 25m v1.15.3

slave1 ready 2m v1.15.3

slave2 ready 2m v1.15.3

Once you are finished, you can proceed to the next step.

## Deploy Service.jar on the Kubernetes Cluster:-

## **Creating a service for an application running in two pods**

## Run a service .jar application in your cluster

Create a Docker compose file in /home/ubuntu

**Vi /home/ubuntu/service.yaml**

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: service-deployment**

**labels:**

**app: service**

**spec:**

**replicas: 3**

**selector:**

**matchLabels:**

**app: service**

**template:**

**metadata:**

**labels:**

**app: service**

**spec:**

**containers:**

**- name: servicecont**

**image: (image file or we can pull image from Docker hub.,)**

**ports:**

**- containerPort: 8057**

Wq!

**kubectl apply -f /home/ubuntu/service.yaml**

**he preceding command creates a**[**Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)**object and an associated [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) object. The ReplicaSet has two**[**Pods**](https://kubernetes.io/docs/concepts/workloads/pods/pod/)**, each of which runs the service .jar application**

1. **Display information about the Deployment:**

**kubectl get deployments service-deployment**

**kubectl describe deployments service-deployment**

1. **Display information about your ReplicaSet objects:**

**kubectl get replicasets**

**kubectl describe replicasets**

1. **Create a Service object that exposes the deployment:**

**kubectl expose deployment service-deployment --type=LoadBalancer --name=my-servicecont(customize name)**

1. **Display information about the Service:**

**kubectl get services servicecont(customize name)**

**The output is similar to this:**

**NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE**

**servicecontLoadBalancer 10.3.245.137 104.198.205.71 8080/TCP 54s**

**Note: The type=LoadBalancer service is backed by external cloud providers, which is not covered in this example, please refer to**[**this page**](https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer)**for the details.**

**Note: If the external IP address is shown as <pending>, wait for a minute and enter the same command again.**

1. **Display detailed information about the Service:**

**kubectl describe services servicecont**

**The output is similar to this:**

**Name: servicecont**

**Namespace: default**

**Labels: app.kubernetes.io/name=load-balancer-example**

**Annotations: <none>**

**Selector: app.kubernetes.io/name=load-balancer-example**

**Type: LoadBalancer**

**IP: 10.3.245.137**

**LoadBalancer Ingress: 104.198.205.71**

**Port: <unset> 8080/TCP**

**NodePort: <unset> 32377/TCP**

**Endpoints: 10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more...**

**Session Affinity: None**

**Events: <none>**

**Make a note of the external IP address (LoadBalancer Ingress) exposed by your service. In this example, the external IP address is 104.198.205.71. Also note the value of Port and NodePort. In this example, the Port is 8080 and the NodePort is 32377.**

1. **In the preceding output, you can see that the service has several endpoints: 10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more. These are internal addresses of the pods that are running the Service .jar application. To verify these are pod addresses, enter this command:**

**kubectl get pods --output=wide**

**The output is similar to this:**

**NAME ... IP NODE**

**servicecont-2895499144-1jaz9 ... 10.0.1.6 gke-cluster-1-default-pool-e0b8d269-1afc**

**servicecont-2895499144-2e5uh ... 10.0.1.8 gke-cluster-1-default-pool-e0b8d269-1afc**

**servicecont-2895499144-9m4h1 ... 10.0.0.6 gke-cluster-1-default-pool-e0b8d269-5v7a**

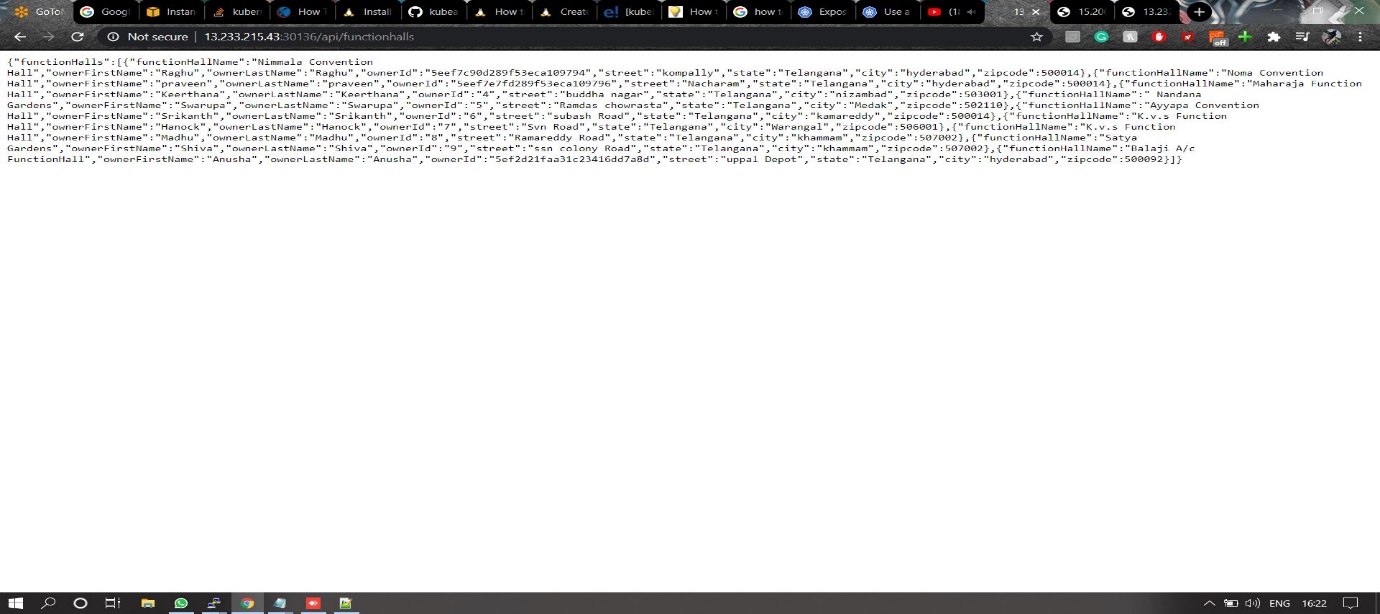
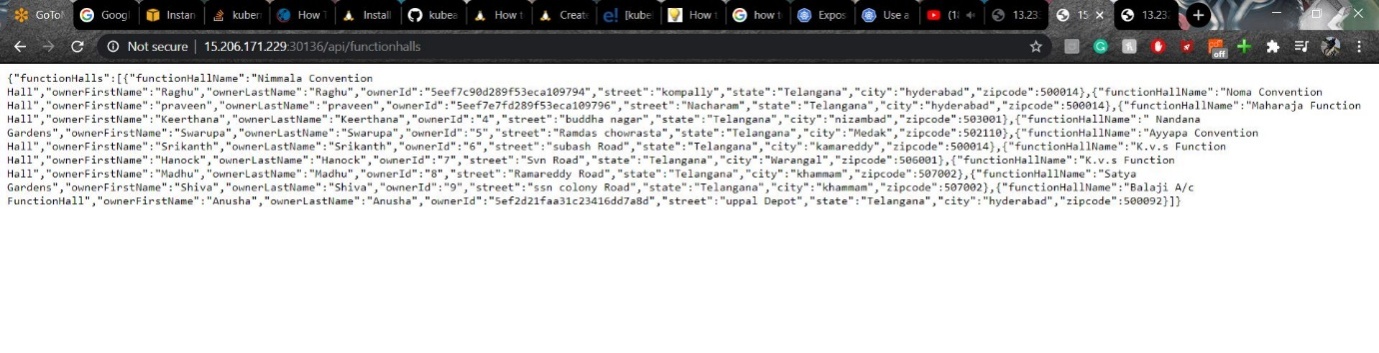
**servicecont-2895499144-o4z13 ... 10.0.1.7 gke-cluster-1-default-pool-e0b8d269-1afc**

**servicecont-2895499144-segjf ... 10.0.2.5 gke-cluster-1-default-pool-e0b8d269-cpuc**

1. **Use the external IP address (LoadBalancer Ingress) to access the Service .jar application:**

**curl http://<public-ip>:<port>**

**where <public-ip> is the external IP address (LoadBalancer Ingress) of your Service, and <port> is the value of Port in your Service description. If you are using minikube, typing minikube service servicecont will automatically open the Service .jar application in a browser.**

**The response to a successful request is a hello message:**

**Reference Screenshots:-**

