Beginners guide to learn Kubernetes Architecture

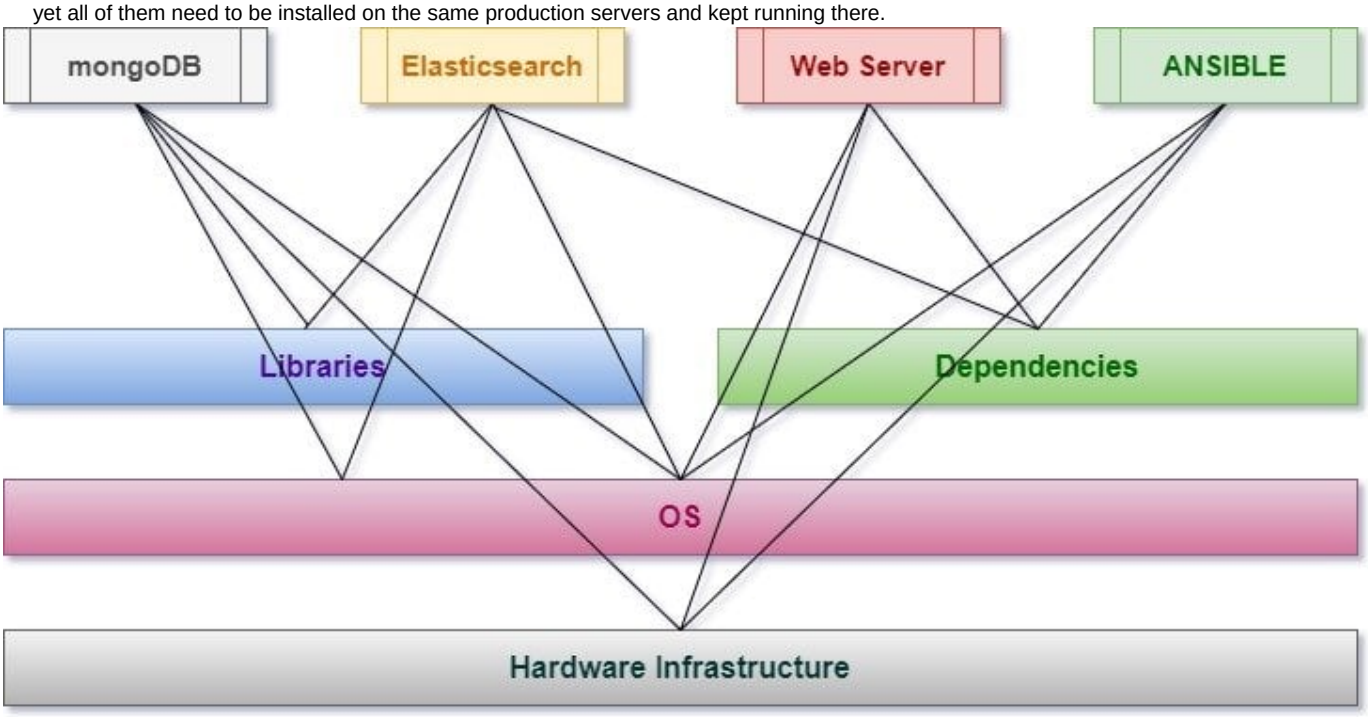
Table of Contents

Kubernetes is also known as K8s was created by Google based on their experience with containers in production. It is an open-source project and one of the best and most popular container orchestration technology.

Before we understand Kubernetes, we must be familiar with Docker containers, or else if you are a beginner then you may find Kubernetes a little confusing to understand. So we will start our Kubernetes Tutorial by first understanding Dockers and Containers.

What are Docker containers and why you should use them?

- In a normal IT, workflow developers would develop a new application. Once the application development was completed, they would hand over that application to the operations engineers, who were then supposed to install it on the production servers and get it running.
- If the operations engineers were lucky, they even got a somewhat accurate document with installation instructions from the developers. So far, so good, and life was easy.
- But things get a bit out of hand when, in an enterprise, there are many teams of developers that create quite different types of applications, yet all of them need to be installed on the same production servers and kept running there.

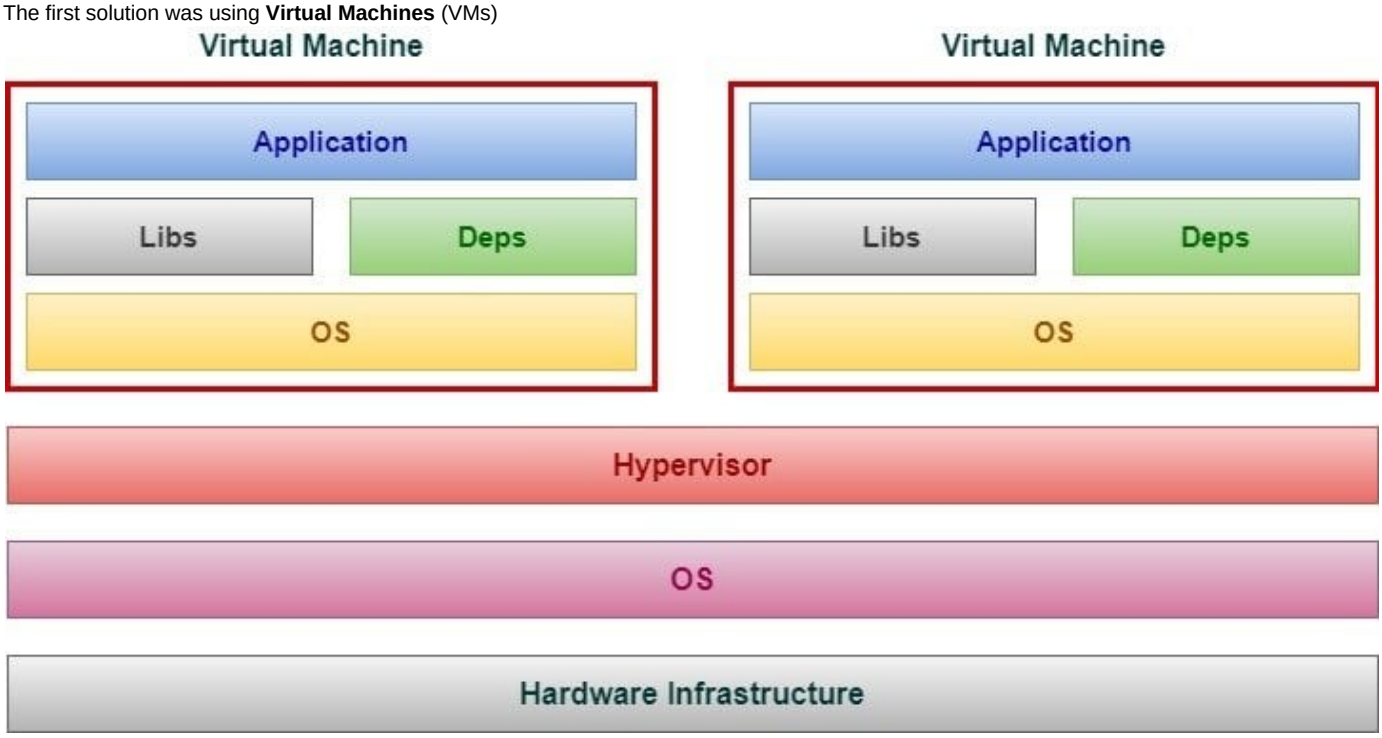


- Usually, each application has some external dependencies, such as which framework it was built on, what libraries it uses, and so on. Sometimes, two applications use the same framework but in different versions that might or might not be compatible with each other.
- So, installing a new version of a certain application can be a complex project on its own, and often needed months of planning and testing

- But these days we have to release a patch, update very often so this development and testing cycle **can be very risky** to the business.

Using Virtual Machine

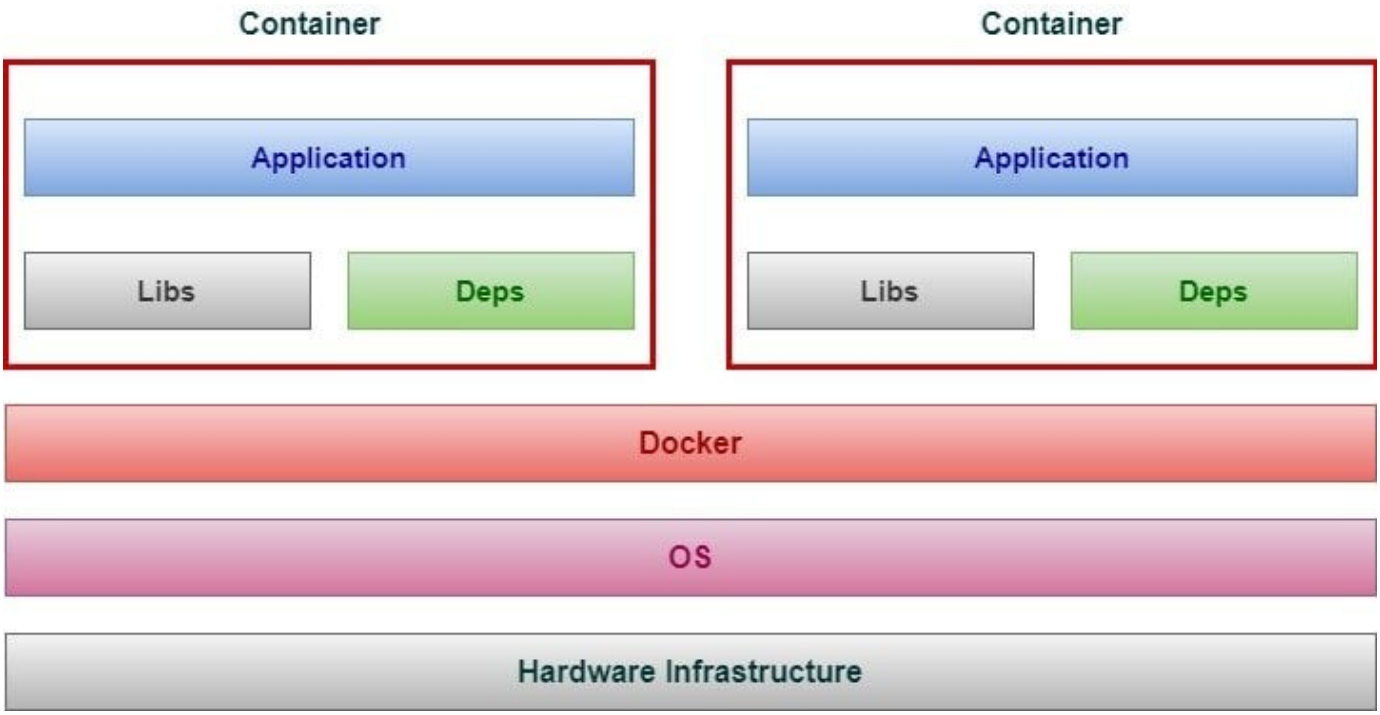
The first solution was using **Virtual Machines** (VMs)



- Instead of running multiple applications, all on the same server, companies would package and **run a single application on each VM**
- With this, all the compatibility problems were gone and life seemed to be good again.
- But this comes with its own set of demerits where each VM needs a lot of resources where most is used by underlying system OS.

Using Docker containers

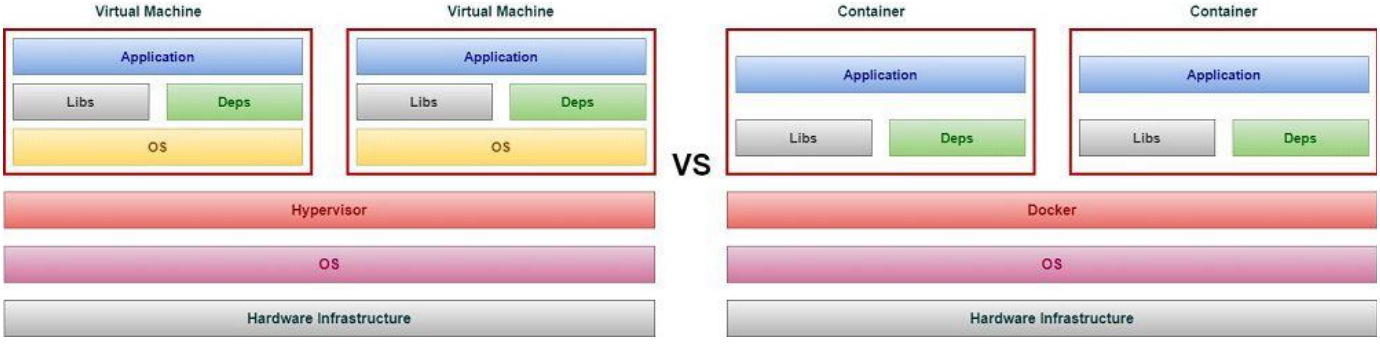
The ultimate solution to this problem was to provide something that is much more lightweight than VMs - **Docker container** to the rescue.



- Instead of virtualizing hardware, containers rest on top of a single Linux instance. This allows Docker to leave behind a lot of bloat associated with a full hardware hypervisor.
- **Don't mistake** Docker Engine (or the LXC process) as the equivalent of a hypervisor in a more traditional VM, it is simply encapsulating process on the underlying system.
- Docker utilizes the **namespace** feature of the Linux kernel wherein the namespaces will make the processes that are running within one container are invisible for processors, or users running within another container
- With docker, developers would now package their application, dependent libraries, framework in a container to the testers or operation engineer.
- To testers and operations engineers, a container is just a black box, and all they need is a Linux OS with Docker running and they can easily deploy the container without having to worry about configuring the application as these containers already contain an up and running application.

Virtual Machine vs Docker Containers

The image should be self-explanatory to understand the difference between **Docker and VMware** architecture.

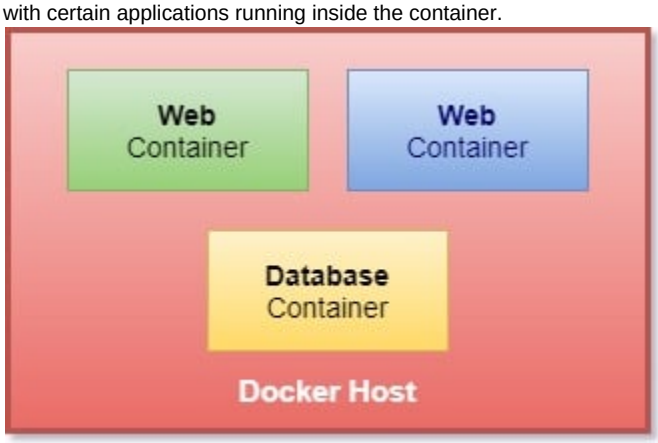


- VM requires a Hypervisor that can be either installed on an operating system or directly on the hardware while a container can be deployed after installing docker.
- VM requires a separating OS to be installed to deploy your application while Docker containers share the host operating system, and that is why they are lightweight
- Since Docker shares OS with the host, the bootup time of the docker container is very less while it is comparatively higher for VMs
- The docker containers share Linux kernel so it would be a good fit if you are planning to run multiple applications on the same Linux kernel but if you have applications that require a different operating system then you will have to go for VM
- Since VM does not share the host OS it is comparatively more secure than Docker containers. An attacker may exploit all the containers if it gets access to the host or any single container.

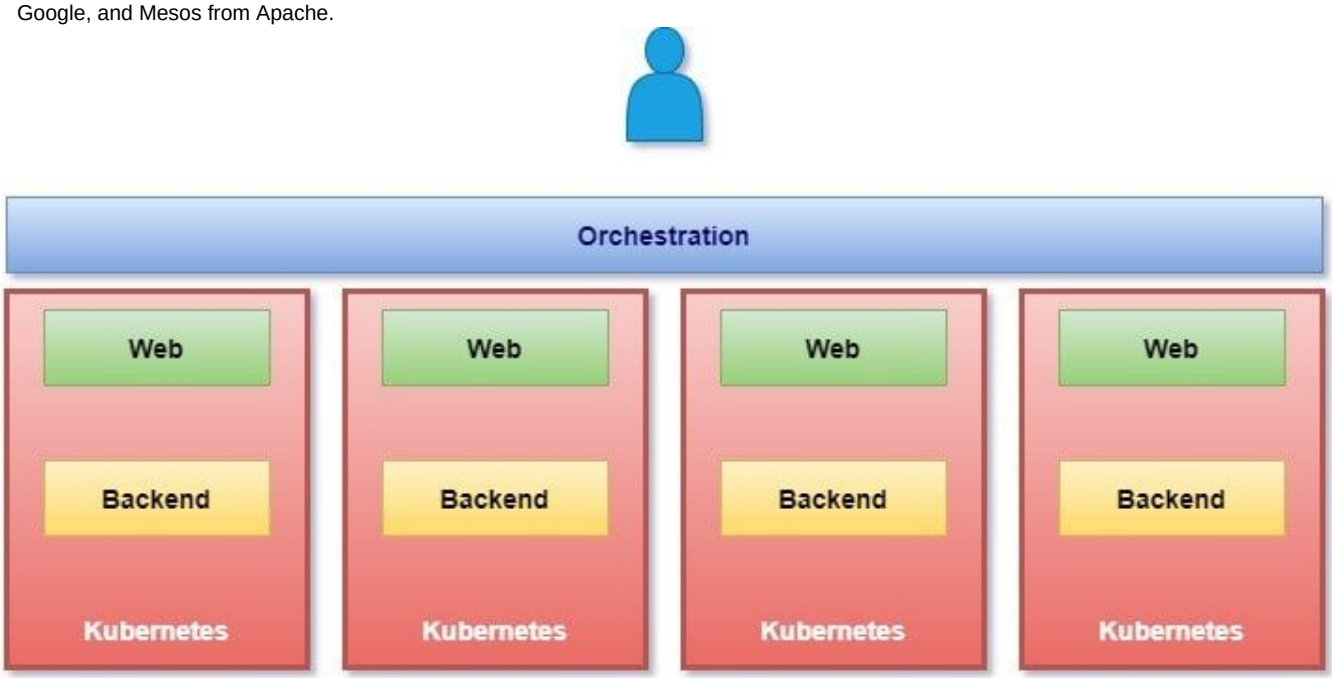
- Since containers don't have OS they use comparatively very few resources to execute the application and you can utilize the underlying resources more effectively

Container Orchestration

Now that you are familiar with containers, next, we need to learn about **container orchestration**. Just to summarise we have a docker container with certain applications running inside the container.



- It is possible that your application from container 1 is dependent on some other application from another container such as database, message, logging service in the production environment.
- You may also need the ability to scale up the number of containers during peak time, for example, I am sure you must be familiar with Amazon sales during holidays when they have a bunch of extra offers on all products. In such case, they need to **scale up** their resources for applications to be able to handle more users. Once the festive offer is finished then they would again need to **scale down** the number of containers with applications.
- To enable this functionality we need an underlying platform with a set of resources and capabilities. The platform needs to **orchestrate** the connectivity between the containers and automatically scale up or down based on the load.
- This whole process of deploying and managing containers is known as **container orchestration**
- **Kubernetes is thus a container orchestration technology** used to orchestrate the deployment and management of hundreds and thousands of containers in a cluster environment.
- There are multiple similar technologies available today, docker has its own orchestration software i.e. Docker Swarm, Kubernetes from Google, and Mesos from Apache.

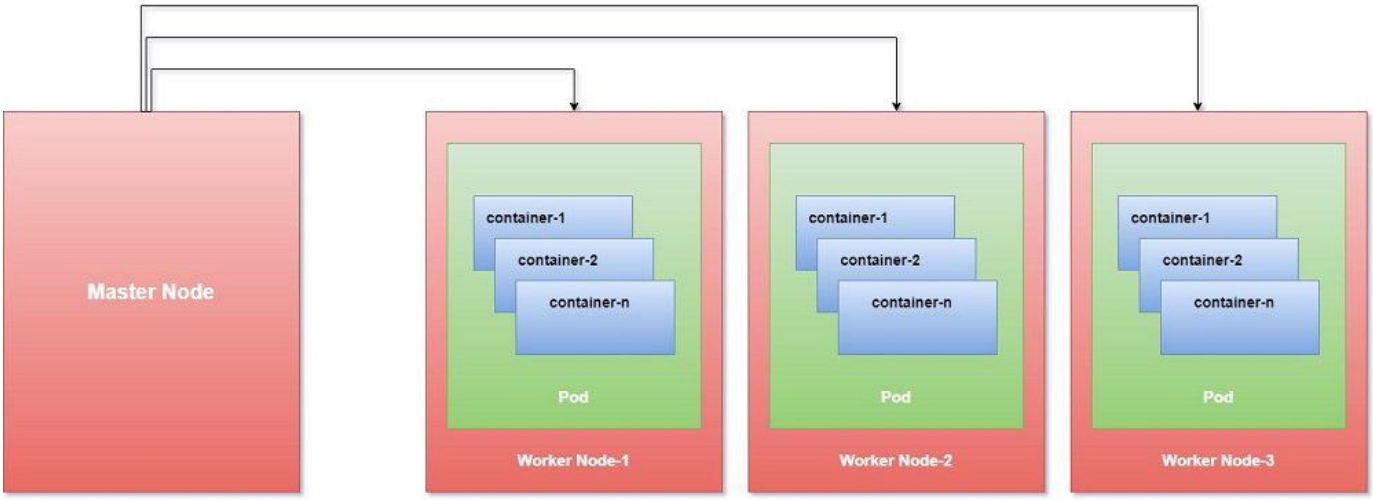


- Your application is now **highly available** as now we have multiple instances of your application across multiple nodes
- The user traffic is load balanced across various containers

- When demand increases deploy more instances of the applications seamlessly and within a matter of seconds and we have the ability to do that at a service level when we run out of hardware resources then scale the number of underlying nodes up and down without taking down the application and this all can be done easily using a set of the declarative object configuration file.

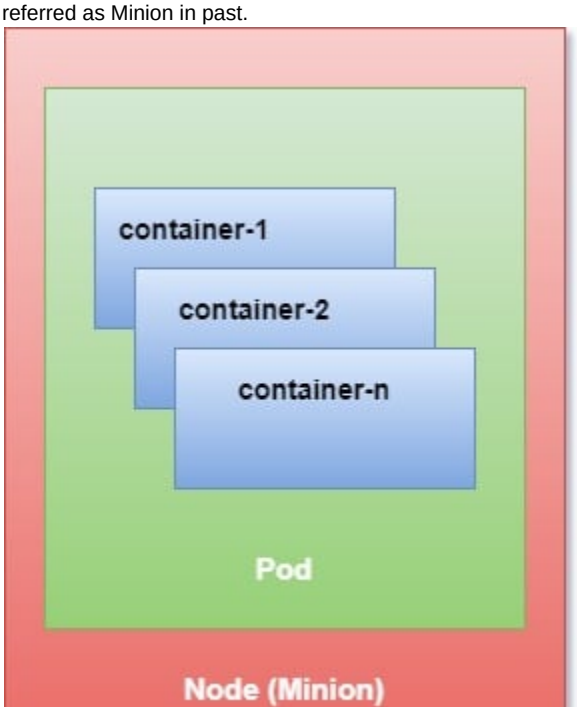
Kubernetes Architecture

- A **Kubernetes Cluster** consists of a Master and Client node setup where we will have one Master or **Controller** node along with multiple Client nodes also referred to as **worker** nodes or in minions.
- A **Master** is a node with Kubernetes installed and is responsible for the actual orchestration of containers on the worker nodes. It will contain all the information of cluster nodes, monitor each node and if a worker node fails then it will move the workload from the failed node to another worker node.



- A **Node** is a worker machine that can be a physical or virtual machine on which Kubernetes is installed.
- Kubernetes does not deploy containers directly into the worker nodes, the containers are encapsulated into a Kubernetes object known as **Pods**.
- A pod is a single instance of an application and they are the smallest deployable units of computing that you can create and manage in Kubernetes.
- You will deploy containers inside these pods where you will deploy your application

Following is a basic architectural diagram of Kubernetes and the relationship between containers, pods, and physical worker nodes which were referred as Minion in past.



Kubernetes Cluster

There are three main components in the Kubernetes Cluster i.e. Nodes, Pods and Containers. We have already learned about containers in depth so I will cover the remaining components here:

Master

- The master is the control plane of Kubernetes.
- It consists of several components, such as an API server, a scheduler, and a controller manager.
- The master is responsible for the global state of the cluster, cluster-level scheduling of pods, and handling of events. Usually, all the master components are set up on a single host.
- When considering high-availability scenarios or very large clusters, you will want to have master redundancy.

Nodes (Minion)

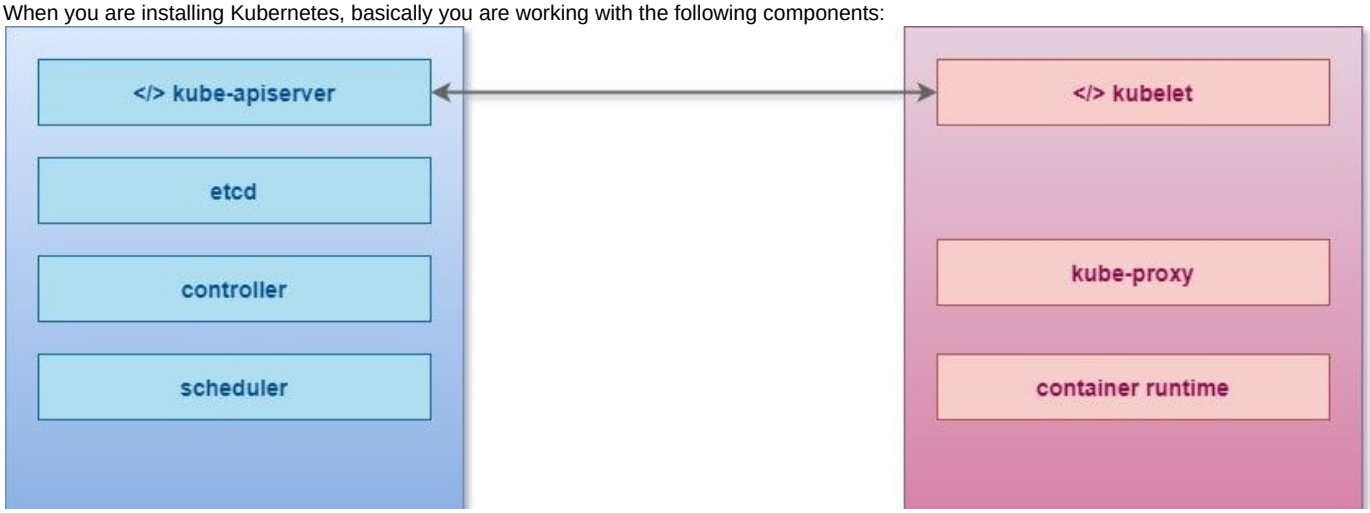
- You can think of these as container clients.
- These are individual hosts (physical or virtual) on which Docker would be installed to host different containers within your managed cluster
- Each Node will run ETCD (keypair management and communication service, used by Kubernetes for exchanging messages and reporting Cluster status) as well as the Kubernetes proxy

Pods

- A Pod is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers.
- We're not implying that a pod always includes more than one container—it's common for pods to contain only a single container.
- The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node—it never spans multiple worker nodes
- These containers are guaranteed (by the cluster controller) to be located on the same host machine in order to facilitate sharing of resources
- Pods are assigned a unique IP address within each cluster. These allow an application to use ports without having to worry about conflicting port utilization
- Pods can contain definitions of disk volumes or share and then provide access from those to all the members (containers) with the pod.

Kubernetes Components

When you are installing Kubernetes, basically you are working with the following components:



API Server

- The API server acts as a frontend for Kubernetes
- It can easily scale horizontally as it is stateless and stores all the data in the etcd cluster.
- The users, management devices, command line interfaces, all talk to API server to interact with the Kubernetes cluster

etcd key store

- It is a distributed reliable key value store
- Kubernetes uses it to store the entire cluster state
- In a small, transient cluster a single instance of etcd can run on the same node with all the other master components.
- But for more substantial clusters, it is typical to have a three-node or even five-node etcd cluster for redundancy and high availability.
- It is responsible for implementing locks within the clusters to ensure that there are no conflicts between the masters

Scheduler

Kube-scheduler is responsible for scheduling pods into nodes. This is a very complicated task as it needs to consider multiple interacting factors, such as the following:

- Resource requirements
- Service requirements
- Hardware/software policy constraints
- Node affinity and anti-affinity specifications
- Pod affinity and anti-affinity specifications
- Taints and tolerations
- Data locality
- Deadlines

Controllers

- The controllers are the brain behind Orchestration.
- They are responsible for noticing and responding when Nodes, containers, or endpoints goes down
- The controller makes the decision to bring up new containers in such a case

Controller Runtime

It is the underlying software that is used to run containers. In our case, we will be using Docker as the underlying container but there are other options as well such as:

- Docker (via a CRI shim)
- rkt (direct integration to be replaced with Rktlet)
- CRI-O
- Frakti (Kubernetes on the Hypervisor, previously Hypernetes)
- rktlet (CRI implementation for rkt)
- CRI-containerd

The major design policy is that Kubernetes itself should be completely decoupled from specific runtimes. The **Container Runtime Interface (CRI)** enables it.

kubelet

It is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected. That includes the following:

- Receiving pod specs
- Downloading pod secrets from the API server
- Mounting volumes
- Running the pod's containers (via the configured runtime)
- Reporting the status of the node and each pod
- Running the container startup, liveness, and readiness probes

kube-proxy

- Kube-proxy does low-level network housekeeping on each node
- Containers run on the server nodes, but they interact with each other as they are running in a unified networking setup.
- kube-proxy makes it possible for containers to communicate, although they are running on different nodes.
- It reflects the Kubernetes services locally and can perform TCP and UDP forwarding.
- It finds cluster IPs via environment variables or DNS.

Conclusion

In this Kubernetes tutorial, we learned how applications have changed in recent years and how they can now be harder to deploy and manage. You've learned that:

- Linux containers provide much the same benefits as virtual machines, but are far more lightweight and allow for much better hardware utilization.
- Docker improved on existing Linux container technologies by allowing easier and faster provisioning of containerized apps together with their OS environments.
- Kubernetes exposes the whole data center as a single computational resource for running applications.
- Developers can deploy apps through Kubernetes without assistance from sysadmins.
- Sysadmins can sleep better by having Kubernetes deal with failed nodes automatically.