# Kubernetes Notes

Iñaki Lakunza

May 28, 2024

## Contents

# 1 Introduction

Kubernetes is an open source container orchestration framework, originally developed by Google.

It manages containers, be Docker containers or from some other technology.

This means that Kubernetes helps manage applications that are made up of a large number of containers, helping manage them in different environments, such as physical machines, virtual machines or cloud environments, or even hybrid deployment environments.

**What problems does Kubernetes solve?**

The containers offer the perfect host for small independent applications like microservices, so we need a way o managing a large number of containers across multiple environments.

**What features do orchestration tools offer?**

- High availability or no downtime.
- Scalability or high performance.
- Disaster recovery - backup and restore.

# 2 Kubernetes Components

- **Pod**: The basic component or the smallest unit of Kubernetes is a pod. It is an abstraction over a component.

  What pod does is to create a running environment, such as a docker container. It does this because Kubernetes wants to abstract away the container runtime or container technologies so that we can replace them if we want to. And also because we do not have to directly work with the Docker container, so **we only interact with the Kubernetes layer**.

  So we have an application Pod, which is our own application and, that will maybe use a database pod.

  **A pot is usually meant to run one application container inside of it**, even if it is possible to run more than one in one pod.

  So, in our Node we have until now two pods, one which contains the app and the other one containing the database.

  **Each Pod gets its own IP address, NOT THE COTAINER, but the pod**, and each Pod can communicate with each other using that IP address.

  It is important to take into account that **these IP addresses are EPHIMERAL, which means that they can die very easily**. So, this means that if we for example lose the database container because the container crahshed because the application chrashed inside or because the nodes, the server that we are running them on, run out of resources, the Pod will die.

  If this happens **a new Pod will be created in its place and it will get assigned a new IP address**. This is not convenient if we are communicating with the database using the IP address because now we would have to adjust it every time the Pod restarts.

Because of that, another component of Kubernetes called **Service** is used.

- **Service**: A Service is a STATIC IP address or permanent IP address that can be attached to each Pod.

  So our app Pod will have its own service and the database Pod will have its own service.

  And, **the life cycles of the Pod and the Service are not connected**, so even if a Pod dies and gets replaced, the Service will not change.

  We want our application to be accessible through a browser, and for this we would have to create an external Service.

  So, a **External Service** is a Service which opens the communication from external source.

  But obviously we do not want our database to be open to the public requests. So that is why we have to create something called **Internal Service**.

  A Internal Service is a type of a Service that we specify when creating one.

  When we are doing test, we want our address to be easy to work with, a one which has the port number of the service.

  But, for the final application we want to work with a secure protocol and the domain name (https://my-app.com), so, for that there is another component of Kubernetes called **Ingress**.

  So, instead of Service, the request first goes through Ingress and it does the forwarding then to the service.

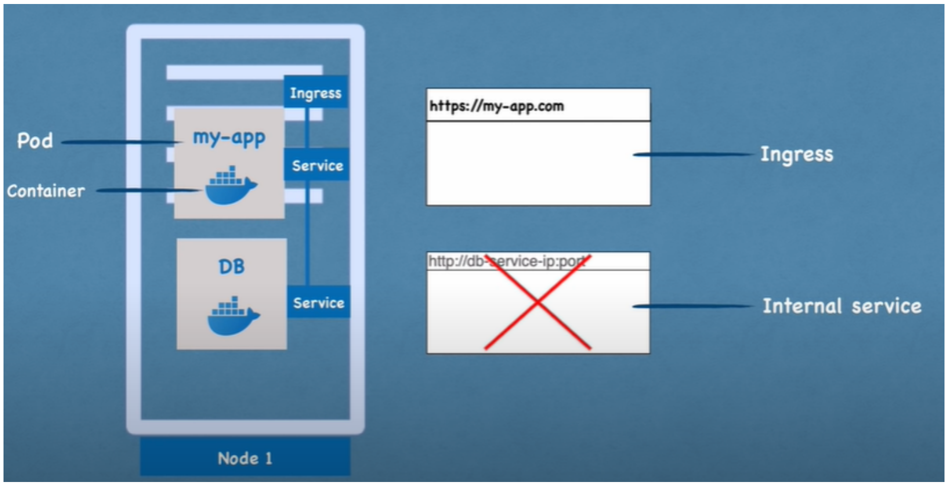


Figure 1:

- **ConfigMap and Secret**: Pods communicate with each other using a Service, so our application will have a database endpoint that it uses to communicate with the database. But where we configurate usually this database URL or endpoint is in application properties file or some kind of external environmental variable. But usually is done inside of the built image of the application.

So, for example, if the endpoint of the Service or Service Name changed, we would have to adjust that URL in the application, so we would have to re-build the application with a new version and we would have to push it to the repository, and now we would have to pull that new image to our Pod and restart everything.

So this is a bit tedious for a very small change like the database URL. For this purpose, Kubernetes has a component called **ConfigMap**.

What is does is our external configuration to our application. So the ConfigMap would usually contain configuration data like URLs of the database or some other services that we use.

And in Kubernetes we just connect it to the Pod, so that the Pod gets the data that the ConfigMap contains (refers to).

And now, if we change the name of the Service, the endpoint of the Service, we just have to adjust the ConfigMap and that's it, we do not have to build a new image and go through the whole cycle.

Part of the external configuration can also be the database username and password, which may also change in the application deployment process. But putting a password or other credentials in a ConfigMap, in a plain text format would be insecure, even though it is an external configuration. So, for this purpose, Kubernetes has another component called **Secret**.

**Secret** is just like a ConfigMap but the difference is that it is used to store secret data credentials, for example. And they are not stored in plain text format, they are stored in *base64 encoding* format.

So, the Secret would contain things like credentials. And just like the ConfigMap, we connect it to the Pod so that it will be able to see and work with the stored information in the Secret.

We can actually use the data from ConfigMap or Secret using **Environmental Variables**, or even as a properties file.

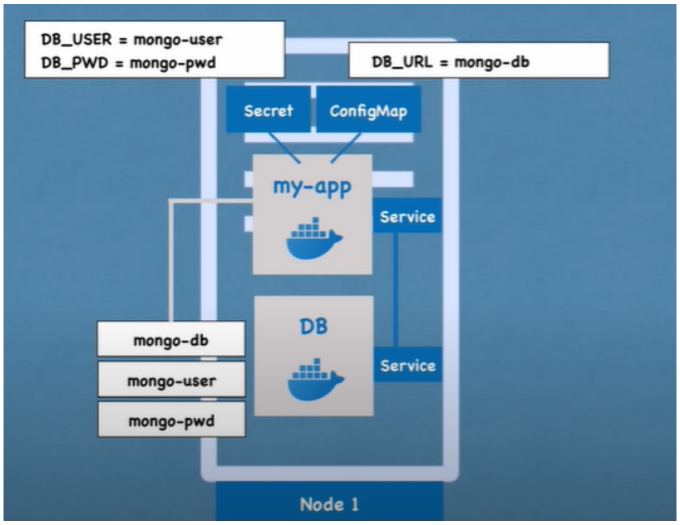


Figure 2:

- **Volumes**: Data Storage is a very important concept we will work with. In our example we have our Database Pod that our application uses and it has some data.

  If the database container or the Pod gets restarted the data would be gone. And we want our database data or log data to be persisted reliably in a long term. So, the way we can do this is by using **Volumes**.

  **Volumes** work by attaching a physical storage on a hard drive to our Pod, and this storage could either be on our local machine (so it would be on the same server node where the Pod is running), or it could also be in a remote storage (meaning that it would be outside the Kubernetes cluster, like a cloud storage or our own premise storage, which is not part of the Kubernetes cluster, so we just have an external reference on it).

  So now, by using the Volume we would be able to keep our data even if the Pod gets restarted.

  **The Kubernetes cluster does not manage any data persistance, so that means that we, as a Kubernetes user, are responsible for backing up the data and replicating and managing it, and making sure that it is kept on a proper hardware, ....**

- **Deployments and StatefulSets**: If we have our application restarted or it crashes it would mean that we would have a downtime where a user will not be able to reach the application.

  So, instead of relying of one application Pod and one database pod we will **replicate everything in multiple servers**.

  **So we would have another node where a replica or clone of our application would run, which will also be connected to the Service.**

  We have said that the Service is like a persistent IP address with a DNS name, so that we do not have to constantly adjust the endpoint when a Pod dies.

  **A Service is also a load balancer**, which means that the service will catch the request and forward it to the Pod that is less busy. SO the service has both of these functionalities.

  But, **in order to create a second replica of our application we would not create a second Pod, we would instead define a BLUEPRINT for our application Pod and specify how many replicas of that Pod we would like to run**.

  This component is known as **Deployment**, and in practice we would not be working with Pods, we would be creating Deployments, because there we can specify how many replicas we want, and we can also scale up or down the number of replicas or Pods we need.

  So, we have said that a Pod is a layer of abstraction on top of the containers and the Deployment is another abstraction on top of Pods, which makes it more convenient to interact with the Pods, to replicate them and configurate them.

  So in practice we would mostly work with Deployments and not with Pods.

  On the other hand, if our Database would die we could not work with it, so we would need another replica. **But we can't replica a database using a Deployment** because the database has a state which is its data, meaning that if we have clones or replicas of the database they would all need to access the same shared data storage. And there we would need some kind of mechanism that manages which Pods are currently writing to that storage or which Pods are reading from that storage in order to avoid data inconsistencies.

This feature is offered by another component called **StatefulSet**, which is specifically meant for applications like databases. So **we should work with StatefulSets when we work with databases, and not with Deployments**,

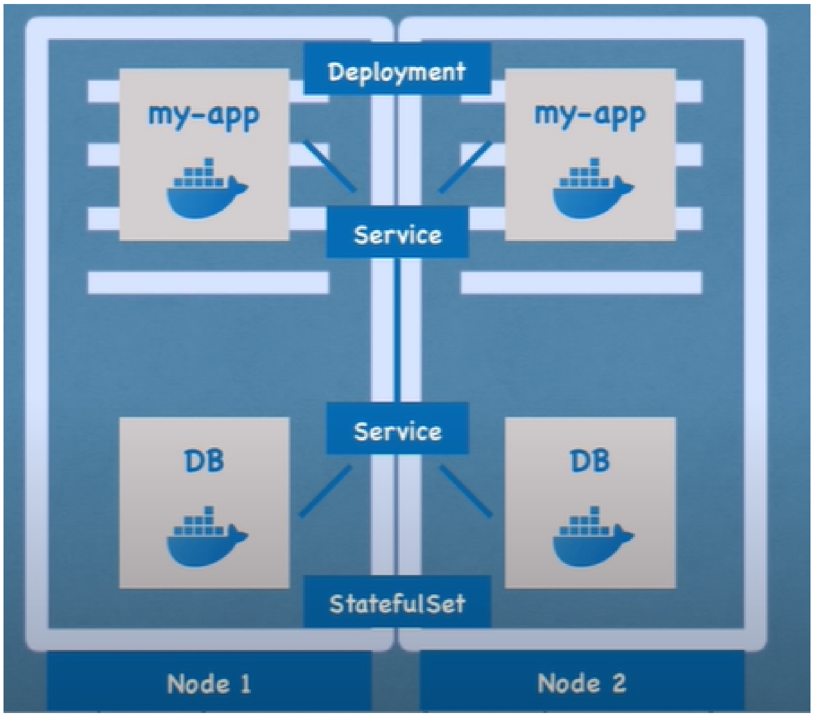


Figure 3:

But we have to take into account that Deploying StatefulSets is not easy. So, it **is common practice to host database applications outside of the Kubernetes cluster and just have the deployments or stateless applications that replicate and scale with no problem inside the Kubernetes cluster, and communicate with the external database**.

# 3 Kubernetes Architecture

We will again use the previous example, where in a node we have an application and a database.

We know that each node will have multiple applications Pods running on that node and the way Kubernetes does it is by using 3 processes that must be installed in every node that are used to schedule and manage those parts.

So nodes are the cluster servers that actually do that work, so that is why they are sometimes called **worker nodes**.

The first process that needs to run every node is the **Container Runtime**. Beacause application Pods have containers running inside Container Runtime needs to be installed on every node. But the process that actually schedules those Pods and the containers that are on it is named **Kubelet**, which is a process of Kubernetes itself unlike Container Runtime that has Interfece with both Container Runtime and the machine, the node itself.

At the end of the day **Kubelet is responsible for taking that configuration and running a Pod or starting a Pod with a container inside, and then assigning resources from the node to the container**. like the CPU RAM storage resources.

Usually a Kubernetes cluster is made up of multiple nodes which also must have Container Runtime and Kubelet services installed. And we can have hundreds of those worker nodes which will run other Pods and containers and replicas of the existing Pods, like in our example the application and the database Pods.

And the way that the communication between them works is using Services, that as we saw, are sort of a load balancer that catches the request directed to the Pod of the application and then forwards it to the respective Pod.

And the third process that is responsible for is for forwarding requests from Services to Pods. This is done by **Kube Proxy**. It also must be installed in every node.

**Kube Proxy** has actually an intelligent forwarding logic inside that makes sure the coommunication also works in a performant way with low overhead. For example, if our application replica is making a request to the database instead of the Service just randomly forwarding the request to any replica, it will actually forward it to the replica that is running **on the same node** as the Pod that initiated the request. Thus, avoiding the network overhead of sending the reques to another machine.

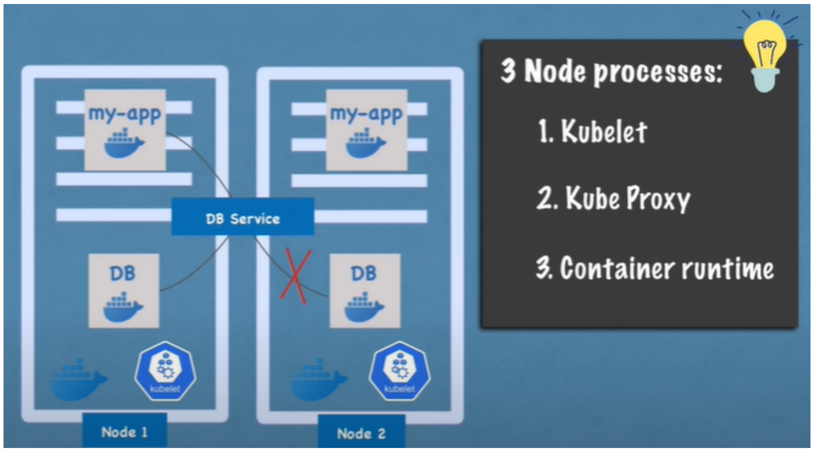
So, to summarize, **Kubelet and Kube Proxy must be installed on every Kubernetes worker node along with an independent Container Runtime, in order for the Kubernetes cluster to function properly**.

**How do we interact with this cluster?**

How do we decide which nodes a new application Pod or Database Pod should be scheduled, or if a replica Pod dies what process actually monitors it and then reschedules it or restarts it.

Or when we add another server how does it join the cluster to become another node and get Pods and other components created on it.

**All these Managing Processes are done by MASTER NODES**



3 Node processes:

1. Kubelet
2. Kube Proxy
3. Container runtime

Figure 4:

## 3.1 Master Processes

Master Servers or Master Nodes have completely different processes running inside. They have 4 processes that run for controlling the cluster state and the worker nodes.

- The first service is **API Service**. When we want to deploy a new application in a Kubernetes cluster we interact with the API server using some client. It could be UI like the Kubernetes dashboard, could be a command line tool like Kubelet or a Kubernetes API.

  So **API Server** is like a cluster gateway, which gets the initial reques of any updates into the cluster or even the queries from the cluster and it also acts as a gatekeeper for authentication to make sure that only authenticated and authorized requests get through the cluster.

  That means whenever we want to schedule new Pods, deploy new applications, create a new service or any other components we have to talk to the API Server on the Master Node and then the API Server will have to validate the request and if everything is fine then it will forward our request to other processes in order to schedule the Pod or create the component we have requested.

  Also if we want to query the status of our deployment or the cluster health, ..., we make a request to the API Server and it will give us the response, which is good for security because we just have one entry point into the cluster.

- **Scheduler**: If we send an API server a request to schedule a new Pod, the API Server after it validates the request it will handle it over to the Scheduler in order to start that application Pod on one of the worker nodes.

  Instead of just randomly assigning to any node, the Scheduler has a intelligent way of deciding in which specific Pod/component it will be scheduled.

  So first it will look at the request and see how much resources the application that we want to schedule will need: how much CPU and RAM; and then it is going to go through the worker nodes and see the available resources for each one of them.

  And it will handle the new Pod on that node.

**The Scheduler just decides on which node the new Pod will be scheduled. The process that actually does the scheduling is the Kubelet**.

- **Controller Manager**: When pods die on any node there must be a way to detect that the nodes died and reschedule those Pods as soon as possible.

  So what the Controller Manager does is detect state changes like crashing of Pods, for instance.

  So, when Pods die, the Controller Manager detects them and tries to recover the cluster state as soon as possible, and for that it makes a request to the Scheduler to reschedule those dead Pods in the same cycle.

  Again, the scheduler makes the decisions based on the resources calculation and makes the reques to the corresponding Kubelets on those worker nodes to restart them.

- **ETCD**: The etcd is a key-value store of a cluster state. We can think of it as a cluster brain.

  Every change in the cluster, such as a new Pod scheduling or the death of a Pod for instance, will get saved or updated into this key-value store.

  All the other processes are based on the information given by the etcd.

  **What is not stored in the etcd key-value store is the actual application data**

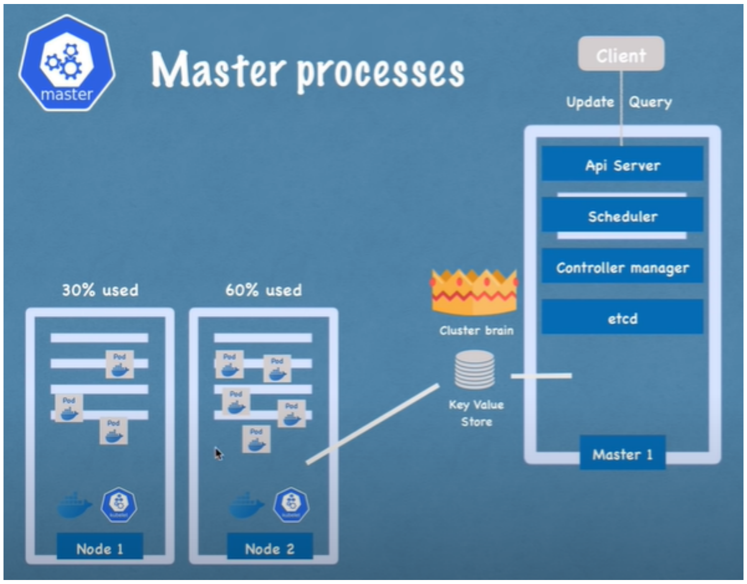


Figure 5:

In practice the Kubernetes cluster is usually made up of multiple Master nodes, where each master node runs its master processes.

The API Server is load balanced, it is distributed accros different master nodes.

**Usually people uses 1 master node for 2 worker nodes, but it depends on the needs.**

# 4 Minikube and Kubectl - Local Setup

Usually, in a Kubernetes world, when we are setting up a production cluster we will have multiple master nodes, at least 2 in a production setting and we would have multiple worker nodes.

And as we know master nodes and worker nodes have their own separate responsibilities.

If we want to test something rapidly in our local machine, but it is untractable to reproduce all the setting in just one local machine.

So that is why there exists **Minikube**. It is a 1 node cluster where the master processes and worker processes both run on 1 node. And this node will have a Docker Container Runtime pre-installed, so we will be able to run the containers or the Pods with the containers on this node.

And the way it is going to run on our laptop is through a virtual box or some other hypervisor.

So, basically, Minikube will create a virtual box on our laptop and the nodes will run there. **We can test Kubernetes this way in our local setup**.

So, once we have a mincluster on our local machine, we need some way to interact with the cluster (create components, configurate, ...), this is where we use **Kubectl**.

**KUBECTL**

Now that we have our virtual node on our local machine that represents a Minikube, we need a way to interact with the cluster, we need a way to create Pods and other Kubernetes components on the node.

**Kubectl** is a command line tool for Kubernetes cluster. One of the master node's processes is the API Server, which is the main entry point into the Kubernetes cluster. So, if we want to do anything with Kubernetes we have to talk to the API Server. We can do this using the UI, the API or the CLI, which is Kubectl.

Once Kubectl submits commands to the API Server the work processes on the Minikube node will make it happen.

**Kubectl is not jsut for the Minikube cluster, if we have a cloud cluster or a hybrid cluster Kubectl is the tool used to interact with the setup**.

Once everything has been installed we can easily create and start the cluster with the following line in a cmd terminal.

We also have to tell Minikube which hypervisor it should use. (In the tutorial she uses **hyperkit** as the hypervisor, but I will use Docker, so, the first line is the one from the tutorial and the second one the one the terminal is telling me to use and the third one the one that I have used)

```
minikube start --vm-driver=hyperkit
minikube start --driver=docker
minikube start --vm-driver=docker
```

Then, we can check the status of the nodes with the following line:

```
kubectl get nodes
```

We can also get the status:

```
minikube status
```

We can also check which version of Kubernetes we have with the following line:

```
kubectl version
```

From now on we will be interacting with the Minikube cluster useing the Kubectl command line tool. So, Minikube is basically for the start up and for deleting the cluster, but everything else is done through Kubectl.

# 5 Main Kubectl Commands

As we said, once we have the cluster set up we are just going to use Kubectl to work with it.

We can check the Pods and the Services with the following lines:

```
kubectl get pod
kubectl get services
```

We will use the **create** command to create components. We can use **kubectl create -h** to see the different options, and as we will see there is no Pod option.

This is because the Pod is the smallest unit, but **in practice we are not working with the Pods directly, we are working with an abstraction layer over the Pods which is called DEPLOYMENT**. So this is what we will be creating.

We have to give a name to the Deployment. And then there are different options, such as the **image** and **dry-run**.

**The Pod needs to be created based on a certain image**.

For the example, we will create a **nginx** Deployment (so it will download the image from DockerHub):

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

So now if we use the line **kubectl get deployment** we will get its information.

We will see that it is not ready. And we can also use **kubectl get pod** to get the Pod information. We will see that we will get one, with the name of our deployment and some random hash afterwards.

And we will see that it says *ContainerCreating* in its status, so it is not ready yet.

But once it has been created we will see how the status changes to *Running*.

Once we create the Deployment it will have all the information (blueprint) for creating the Pod.

We can also get information of the replica set using **kubectl get replicaset**

So, how the layers of abstraction work iw the following:

- The Deployment manages a ReplicaSet
- The ReplicaSet manages all the replicas of the Pod
- The Pod is an abstraction of the container

**Everything below the Deployment is handled by Kubernetes**.

If we want to edit something we will have to go through the Deployment, so, using the name we gave to the deployment previously:

```
kubectl edit deployment nginx-depl
```

And we will get a auto-generated configuration file with default values.

We can scroll down and for instance change the version of the image we are using.

And then, if we try **kubectl get pod** we will see that the previous Pod is being terminating and the new one is running. And same with the ReplicaSet.

### Debugging Pods

We can log to some Pods (for the case of nginx not, but yes with Mongo for instance), for that we have to create the deployment and then log in using the associated Pod name, for instance:

```
kubectl get pod

(it will show the info)

kubectl logs _______
```

We can also use **kubectl exec**, it gets the terminal of the application container we are using (for instance with Mongo):

```
kubectl exec -it _____ -- bin/bash
```

And this way we will access the terminal.

And for exiting we can just use **exit**.

We will want to delete the Pods, so we will have to delete the deployment, using its name (**the name of the deployment, not the Pod**):

```
kubectl delete deployment ____
```

And we will see how the Pod and the replica will be deleted.

If we want to change a large number of configurations the easiest way to go is to create a default Deployment and then to create a yaml file specifying the custom configurations, then we can just use the following line:

```
kubectl apply -f (config_file_name).yaml
```

**The easiest way to do this is by creating a configuration file using as copy the default one and changing it there**, for instance (there we would copy the default file):

```
touch nginx-deployment.yaml
vim nginx-deployment.yaml
```