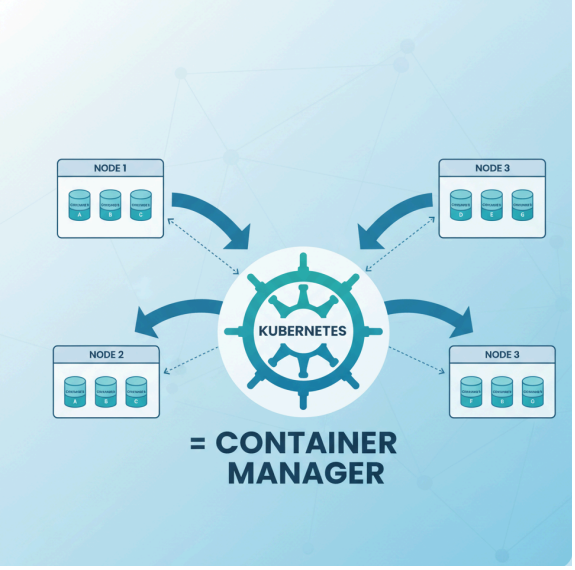
Introduction

What Is Kubernetes

Kubernetes is a container manager. If containers are little boxes that run your applications, Kubernetes is the smart system that decides

where they should run
how many should run
what to do if one breaks
how to keep everything healthy



Short Analogy

Pretend you're running a big delivery company.
Each delivery boy = a container
Your job = send them to the right locations, track them, and replace them if someone quits.

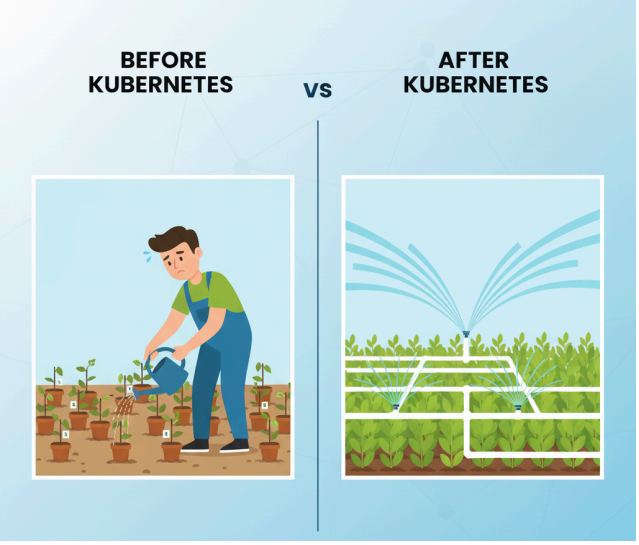
Kubernetes acts like the central manager who does all this automatically.

Why Do We Need Kubernetes?

Running one or two containers is easy.
Running hundreds or thousands is not.

You need a system that

restarts containers if they fail
spreads load across multiple machines
scales automatically when traffic increases
rolls out updates without downtime
keeps track of every running part



Short Analogy

Thinking of watering a single plant is easy.
Now imagine watering 10,000 plants every day.

You'd need an automatic sprinkler system.
Kubernetes=sprinkler system for containers.

Containers vs Pods vs Virtual Machines

Understanding these three is crucial.

Virtual Machines (VMs)

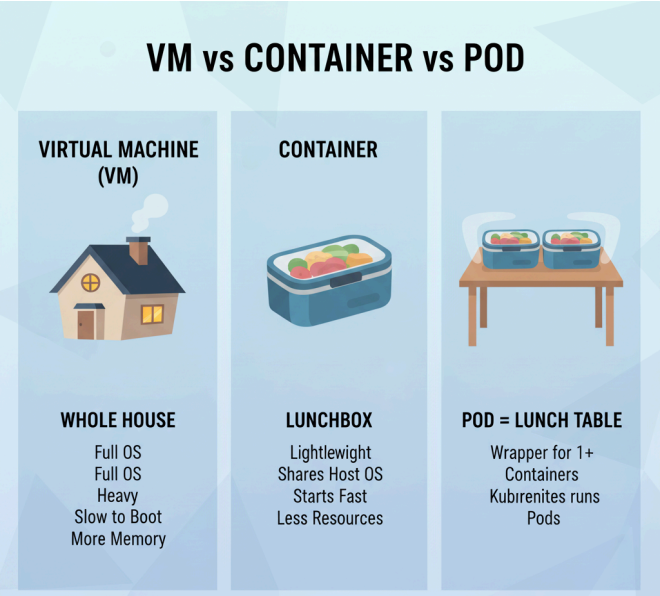
- Include full operating systems
- Heavy
- Slow to boot
- Use more memory

Containers

- Lightweight
- Share the host OS
- Start fast
- Use fewer resources

Pods

- A Pod is a wrapper around one or more containers.
- Kubernetes doesn't run containers directly .. it only runs pods.



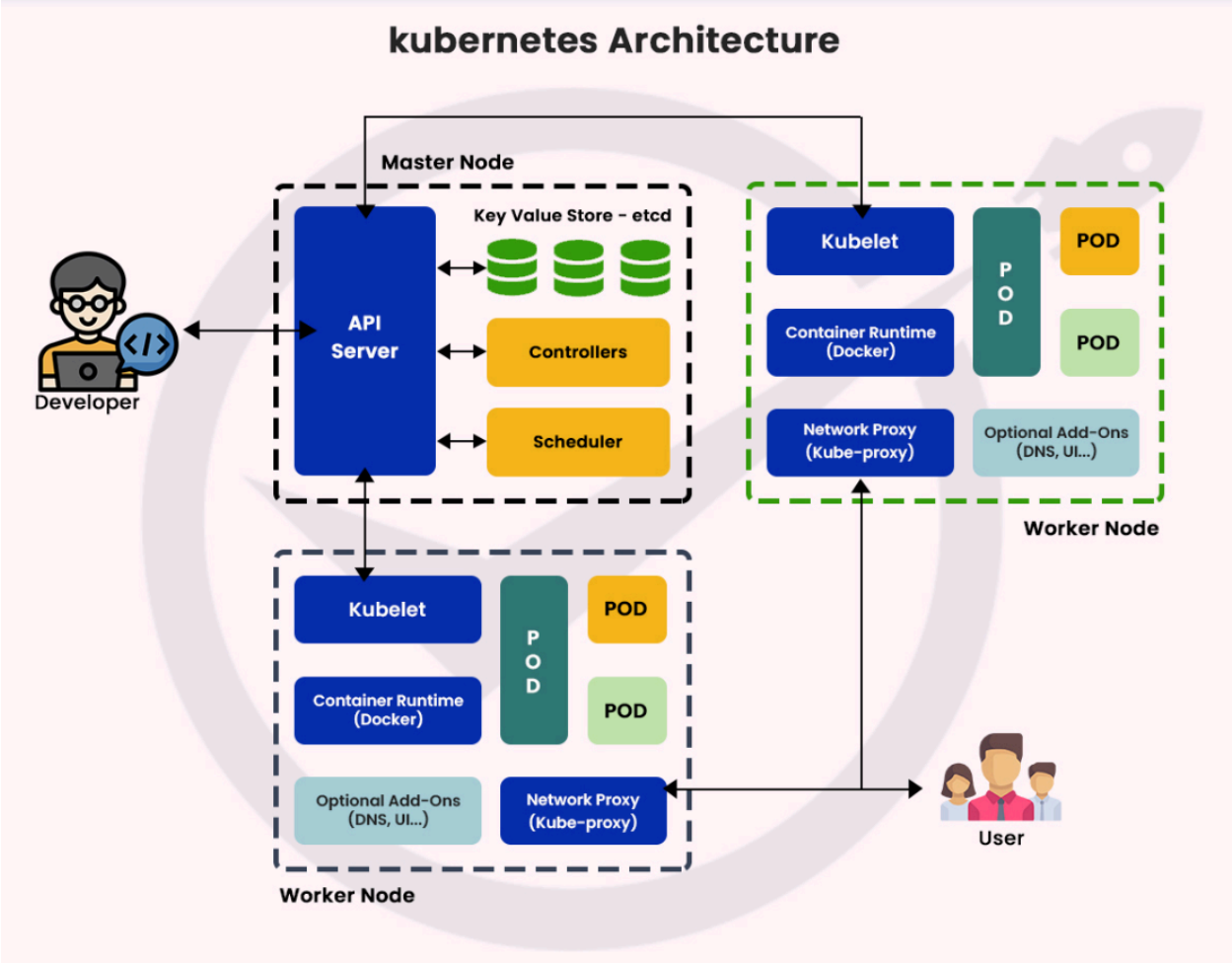
VM: A whole house just for one person

Container: A lunchbox with only the food you need

Pod: A lunch table where 1–2 lunchboxes sit together and share the same space.

Kubernetes Architecture

Before we explain everything, look at this image once.



If this image feels overwhelming- don't worry.
Everyone feels the same the first time.

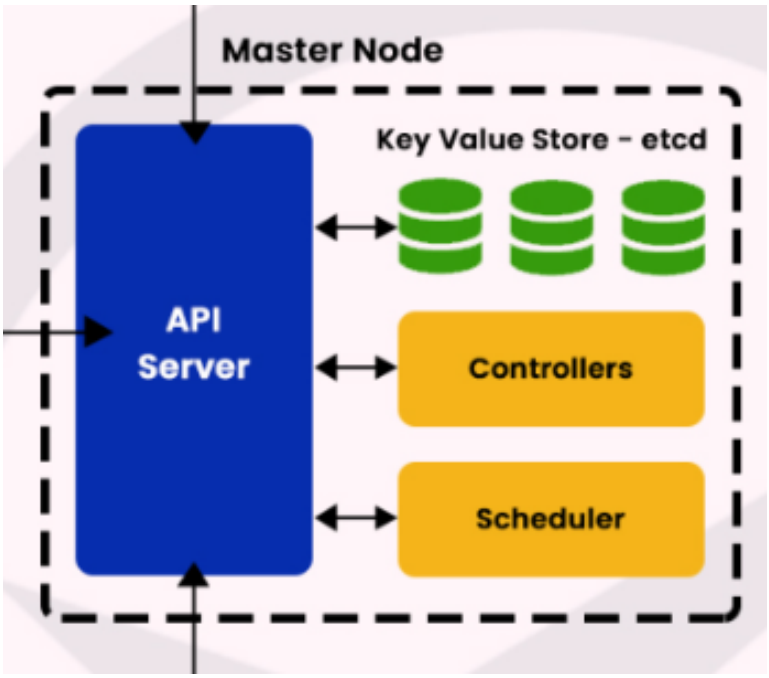
We're going to break it into small, simple pieces using a very easy analogy.
After a few explanations and a bit of practice, this whole architecture will make perfect sense.

Kubernetes Architecture = 3 Simple Sections

Think of Kubernetes like a big, well-run organization.

You can divide it into:

MASTER : The Brain (Control Plane)



This is the boss.
It *does not* run your applications.
It only thinks, decides, plans, and coordinates.

Inside the Control Plane, we have:

API Server: The Messenger

Everyone (you, Kubernetes components, services) talks through this one door.
Nothing bypasses the API Server.

Scheduler: The Task Allocator

Whenever a pod needs a place to run,
the Scheduler decides which worker node is best,
based on CPU, RAM, resource pressure, and rules.

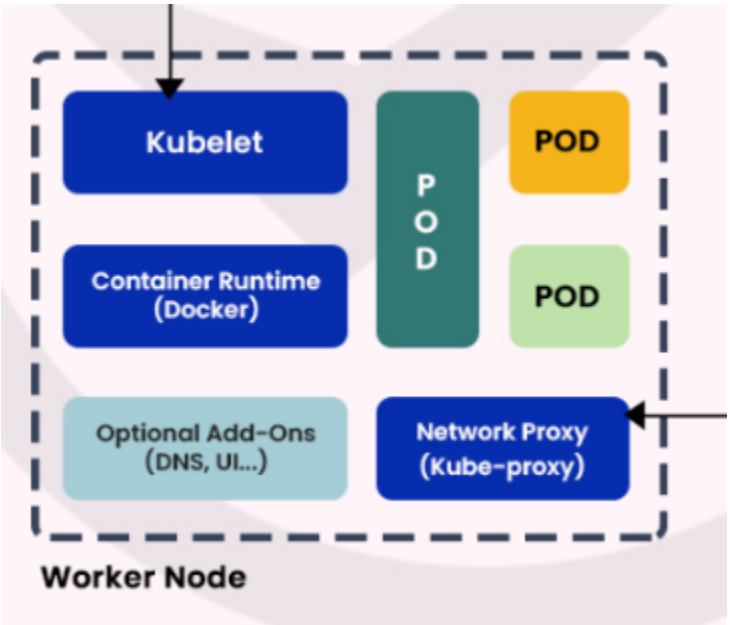
Controller Manager: The Team Leader. It constantly checks:

"Is the cluster healthy?
Are all deployments running correctly?
Are the required number of pods alive?"
If not, it automatically fixes things.

etcd: The Memory of the Cluster.A super reliable key-value database that remembers everything:

- nodes
- pods
- configs
- cluster state

WORKERS The Muscles (Worker Nodes)



These nodes run your actual applications.
Inside each worker node

Kubelet: The Node Manager. This agent talks to the API Server and makes sure:

- pods are running
- containers are healthy
- node status is reported correctly

Container Runtime: The Kitchen. This is where containers actually launch (Docker/containerd).

Pods: The Work Units. Pods contain the containers that run your app code.

Kube-proxy: The network router handles:

- traffic routing inside the cluster
- Load-balancing
- pod-to-pod communication
- service IPs

ACCESS: You & Users

kubectl (Developer Access)
You use kubectl to interact with the cluster.

```
kubectl get pods
kubectl apply -f app.yml
```

Service (User Access)

A Service exposes your application to end users.
Users don't talk to containers, they talk to the Service.

Important Note (Very Common Confusion)
kubectl ≠ kube-proxy

kubectl: Used by *developers / DevOps engineers*
to manage the cluster.

kube-proxy: Used by *users / traffic*
to reach the running application inside pods.

kubectl = management
kube-proxy = networking. And together, all these machines = Your Cluster. Each box in the diagram (master + workers) is a separate:

EC2 instance (AWS),
VM,
or physical machine

When they join together a Kubernetes Cluster is formed. Now that you understand what a cluster is in Kubernetes, let's look at how a cluster is actually created. After that, we'll come back and explain all these components in more detail, one by one.

Kubernetes Cluster (theory)

Before you actually run Kubernetes workloads, you need a cluster. And the good news. There is not just one way to create it there are multiple methods, depending on:

- your experience level
- your machine's resources
- whether you want local learning or cloud-level practice
- cost

To make this simple, think of cluster creation like building a practice ground.

Analogy:
Creating a Kubernetes cluster is like choosing where you want to practice driving:

- A toy car track
- A small training ground
- A professional racing track
- Or actual highways

Each tool matches a different driving environment. Let’s look at all four.

1) **kubeadm (Gives You Full Control)**
Analogy: Driving a manual transmission car. You learn everything: gears, clutch, engine behavior. kubeadm is the tool where:

- *you* install Kubernetes manually
- *you* configure networking
- *you* set up master + worker nodes
- *you* join nodes manually

It is NOT the easiest.
But it teaches you exactly how a real cluster is built on servers.

2) Minikube "Scooter" (Simple, Lightweight, Single Node)
Analogy: A scooter for practicing driving in a small area. Minikube spins up a single-node Kubernetes cluster on your laptop. It handles:

- starting Kubernetes
- Stopping
- enabling addons
- dashboard

It’s easy, friendly, and built for learning but it's a single node, so not ideal for multi-node experience.

3) Kind (Kubernetes in Docker) "RC Remote Control Car Track"

Analogy: A small, perfect, cheap practice track using remote-controlled cars. Kind creates. Kubernetes clusters inside Docker containers. That means:

- No VM required
- No heavy drivers
- Extremely fast to create/delete clusters
- Very low resource usage
- Can create multi-node clusters easily
- Perfect for laptops

4) EKS / AKS / GKE — "Actual Highways" (Real Cloud Clusters)

These are fully-managed Kubernetes clusters from cloud providers:

- EKS → AWS
- AKS → Azure
- GKE → Google Cloud

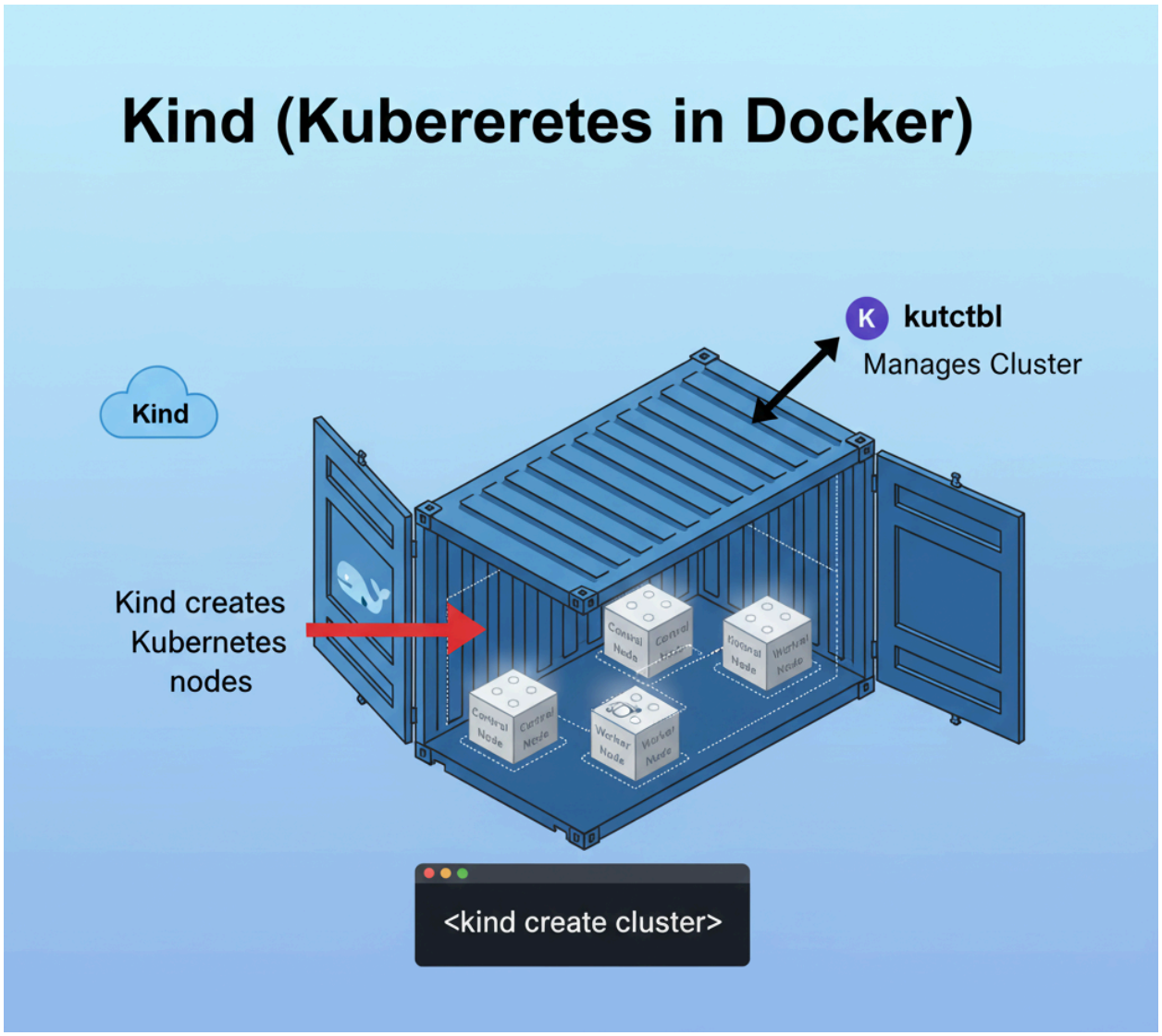
Creating a Kubernetes Cluster (with Kind)

Before running Kubernetes locally, we need three tools:

Docker : used to run containerized nodes

Kind : Kubernetes IN Docker (your cluster runs inside docker containers)

kubectl : CLI tool to interact with Kubernetes



Docker

└── Kubernetes Nodes (created by Kind)

└── Pods / Containers / Services

Where to Set Up Kubernetes (Laptop or Cloud VM? You can say something like:

If your laptop has 8GB RAM, you can run Kind locally.

If you're on Windows Home, or your system is slow, use a cloud VM instead

Minimum VM specs:

2 vCPU

4 GB RAM

20 GB disk

Cloud options: DigitalOcean, AWS EC2, GCP, Azure , etc (your-choice)

Now we are about to see things in action. Here are the steps to create this.

Step 1: Install Docker

Commands:

```
# update your ubuntu system
sudo apt-get update
# install the docker
sudo apt-get install docker.io -y
# configure your user to access the docker commands without sudo in prefix.
# $USER will automatically fetch the current user and this -aG command will append that user in
docker group.
sudo usermod -aG docker $USER && newgrp docker

docker ps
```

If permission denied → log out & log back in

(if still error → reader can check Docker docs)

Step 2: Install Kind (Kubernetes in Docker)

Explanation : Kind is the tool that creates Kubernetes clusters INSIDE Docker containers.

You must install the binary that matches your machine (amd64, arm64, macOS, Linux, etc.).

Here is the official docs: **https://kind.sigs.k8s.io/docs/user/quick-start/#installation**

I am installing it for amd64 (linux)

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.30.0/kind-linux-amd64
# then make it executable
sudo chmod +x ./kind
# now move it to the binaries
sudo mv ./kind /usr/local/bin/kind
```

Step 3: Install kubectl (Kubernetes CLI)

What it is : kubectl is the command-line tool used to talk to your Kubernetes cluster. Without kubectl, you cannot deploy pods, services, or view anything in the cluster.
You must install the binary that matches your machine (amd64, arm64, macOS, Linux, etc.).
Here is the official docs: https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
# this is for verification
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"
# verify
echo "$(cat kubectl.sha256)  kubectl" | sha256sum --check
```

If you see OK, kubectl is authentic and safe to install.

```
ubuntu@ip-172-31-31-163:~$
kubectl: OK
ubuntu@ip-172-31-31-163:~$
```

```
# after the ok output.

sudo chmod +x kubectl
sudo mv ./kubectl /usr/local/bin/kubectl

#verify
kubectl version
```

Creating a Kind Kubernetes Cluster (Using a YAML File)
Kind gives you two ways to create clusters:

1. Using a command
   (quick and simple)

2. Using a YAML file
   (recommended because it gives more control over nodes, networking, ports, etc.)

In real DevOps work, YAML is preferred — it keeps your cluster configuration as code.
So let's start with understanding the syntax of a Kind cluster YAML.

```
kind: <name>
apiVersion: <api-version>
nodes:
  <node> : list
  <extra-configs>
```

kind: Always Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes: A list of nodes you want to create

role: The node type (control-plane or worker)

extraPortMappings: Used for exposing ports to host (example: port 80, 443)

Now Let's Create a YAML for a Real Cluster

Goal:

1 control-plane node + 3 worker nodes,
with port mappings for:
port 80 (HTTP)
port 443 (HTTPS)

Kind documents all versions, node roles, and advanced settings.
https://kind.sigs.k8s.io/docs/user/quick-start/#installation

Create file name kind-cluster.yml and paste this inside that

```
Kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  image: kindest/node:v1.26.0
- role: worker
  image: kindest/node:v1.26.0
- role: worker
  image: kindest/node:v1.26.0
- role: worker
  image: kindest/node:v1.26.0
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 80
    protocol: TCP
```

Why Port Mapping Is Needed?
When Kubernetes runs inside Docker, your cluster is not directly exposed to your host machine.

```
# run
kind create cluster --config=kind-cluster.yml
```

you can verify everything by running:

```
docker ps
```

This shows all Docker containers currently running on your system.
Here's the exact kind of output you should expect:

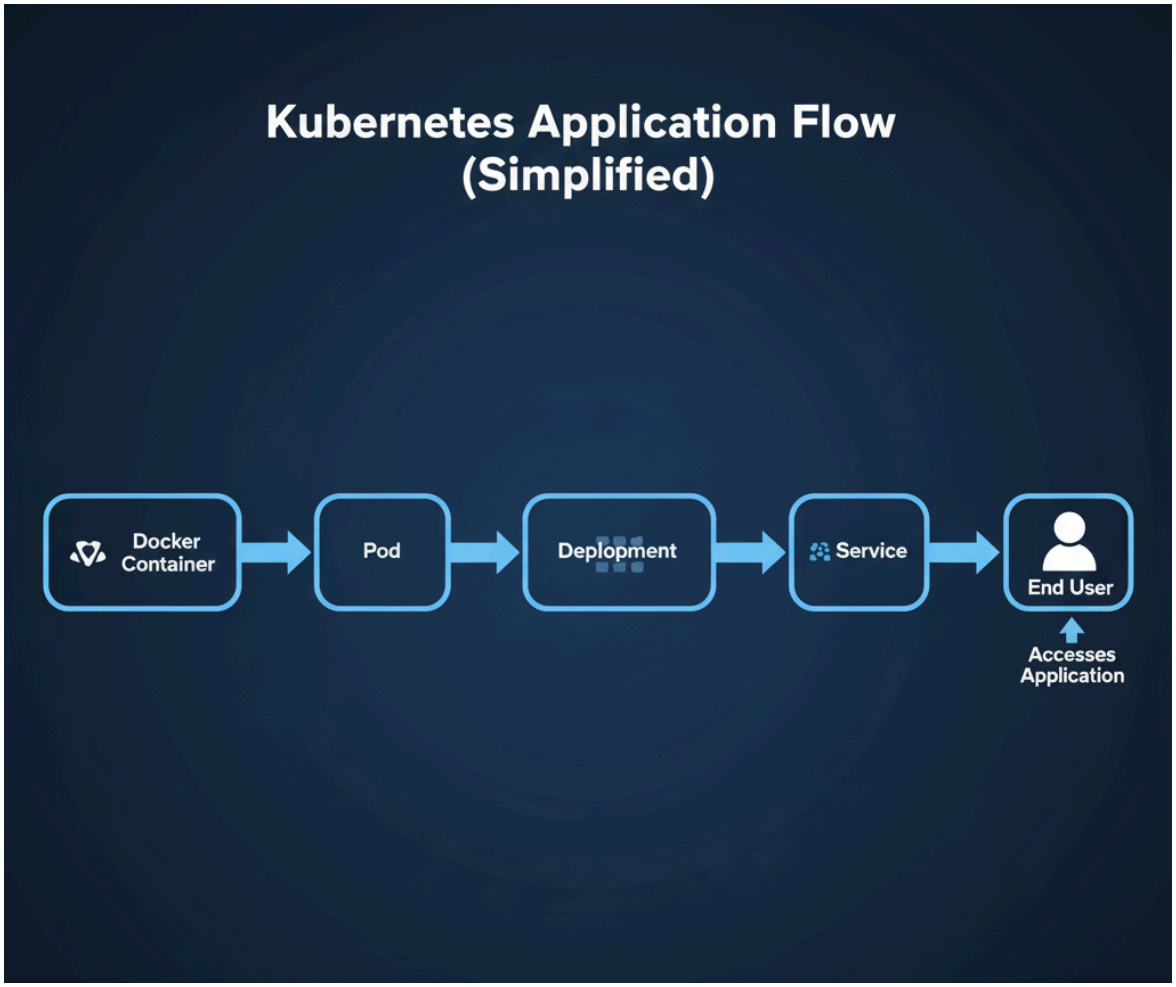
Every line here is not a virtual machine.
Every line is not a bare-metal server.
Every line is a Docker container acting as a Kubernetes node.

- ✔ 4 lightweight Docker containers
- ✔ each container running a full Kubernetes node environment
- ✔ all connected together to form a real Kubernetes cluster

Create Your First Container (NGINX)

In this section, we'll create an NGINX container and run it. Until now, you were probably running containers with a simple **docker run** and doing some port mapping and that was it. With Kubernetes, things work a bit differently.

The image below shows how a user will access your application:



Docker Container: think of this as "your app + its image".

Pod: Kubernetes puts one or more containers *inside* a Pod. So the app actually runs inside a Pod.
Deployment: creates and manages multiple Pods (scaling, rolling updates, etc.).
Service: gives a stable access point (ClusterIP / NodePort / LoadBalancer) so the End User can reach those Pods.

There’s one more important concept you need before we continue: Namespaces.

Namespaces work like separate folders inside your Kubernetes cluster. They keep your resources (Pods, Deployments, Services, etc.) isolated so they don’t mix with resources from other teams or other parts of your application.

Think of it like this:

- You can have nginx running in one namespace
- And mysql running in another namespace
- Even if they both have the same Pod names or Service names, they won’t collide

In the next sections, we’ll create everything using YAML files Namespaces, Pods, Deployments, and Services.

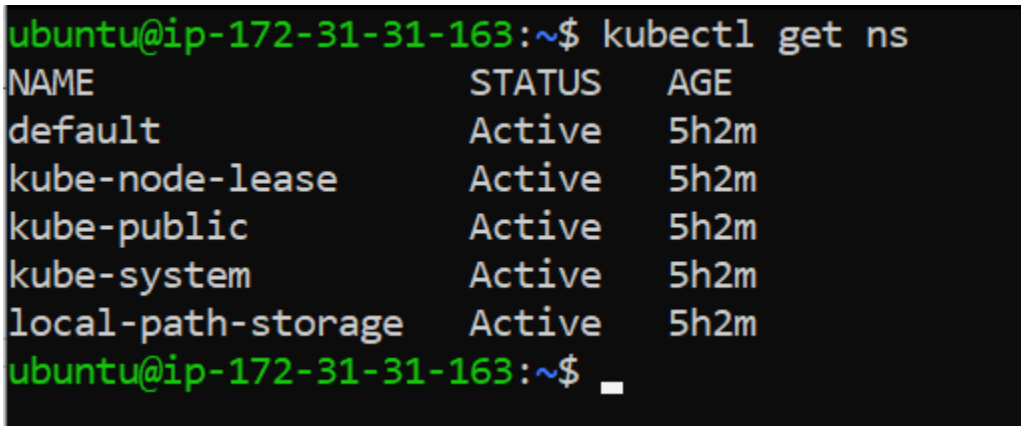
Once you understand how to define these pieces in YAML, the entire Kubernetes workflow will start making sense.

Creating Namespace

Before we start creating our own Namespace, let's take a look at the ones that already exist in your Kubernetes cluster.

Command:

```
kubectl get ns
```



Here's what each one means:

default

The main namespace where everything goes if you don't specify a namespace.
All your first Pods, Deployments, and Services go here by default.

Kube-node-lease

Used internally by Kubernetes.
Stores *heartbeat* information for each node.
Helps Kubernetes detect whether a node is healthy or not.

Kube-public

A namespace that is readable by everyone in the cluster.
Mostly used to store info that needs to be publicly accessible.
Example: cluster information for users.

Kube-system

Contains all system-level components, such as:
kube-dns
kube-proxy

CoreDNS

You should never delete or modify things here unless you truly know what you're doing.

Local-path-storage

Created automatically in Kind or Minikube.
Handles local storage volumes for your Pods.
Useful when you need a PersistentVolume in your local cluster.

Command to create namespace:

```
kubectl create ns nginx
```

for delete

```
kubectl delete ns nginx
```

```
ubuntu@ip-172-31-31-163:~$ kubectl create ns nginx
namespace/nginx created
ubuntu@ip-172-31-31-163:~$ kubectl get ns
NAME                   STATUS   AGE
default                Active   5h11m
kube-node-lease        Active   5h11m
kube-public            Active   5h11m
kube-system            Active   5h11m
local-path-storage     Active   5h11m
nginx                  Active   6s
ubuntu@ip-172-31-31-163:~$
```

Our new namespace is created via command or you can also use yml for that.
Yml to create namespace (optional): create file namespace.yml

```
kind: nameapce
apiVersion: v1
metadata:
  name: nginx
```

Command to run:

```
kubectl apply -f namespace.yml
```

```
ubuntu@ip-172-31-31-163:~$ ubuntu@ip-172-31-31-163:~$ kubectl apply -f namespace.yml
namespace/nginx created
ubuntu@ip-172-31-31-163:~$ kubectl get ns
NAME                 STATUS   AGE
default              Active   5h20m
kube-node-lease      Active   5h20m
kube-public          Active   5h20m
kube-system          Active   5h20m
local-path-storage   Active   5h20m
nginx                Active   10s
ubuntu@ip-172-31-31-163:~$
```

Now, inside this namespace we are going to create pods , deployment and service for our container.

Creating Pod

Now that our cluster is up and running and we understand namespaces, it's time to create our first Pod. In Kubernetes, a Pod is the smallest deployable unit; it runs one or more containers inside it. We'll start simple and run an NGINX container inside a Pod.

Syntax:

```
kind: Pod
apiVersion: v1
metadata:
  name: <pod-name>
  namespace: <namespace>
spec:
  containers:
  - name: <container-name>
    image: <image-name>
    ports:
    - containerPort: <port-number>
```

Actual YAML: NGINX Pod (pod.yml)

```
kind: Pod
apiVersion: v1
metadata:
  name: nginx
  namespace: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

What this YAML means:

kind: Pod: Tells Kubernetes this file defines a Pod
metadata.name: nginx The name of the Pod
metadata.namespace: nginx The Pod will be created inside the nginx namespace
spec.containers: List of containers inside the Pod

image: nginx:latest The container image to run
containerPort: 80 The container listens on port 80 (HTTP)

Command:

```
kubectl apply -f pod.yml
```

```
ubuntu@ip-172-31-31-163:~$ ubuntu@ip-172-31-31-163:~$ kubectl apply -f pod.yml
pod/nginx created
ubuntu@ip-172-31-31-163:~$
```

```
# to verify the command
kubectl get pod -n nginx
```

```
ubuntu@ip-172-31-31-163:~$ kubectl get pod -n nginx
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          37s
ubuntu@ip-172-31-31-163:~$
```

Useful Diagnostic Commands

```
# Here are some helpful commands when working with Pods:
# See all Pods in a namespace
kubectl get pods -n nginx

# Get detailed info about a Pod
kubectl describe pod nginx -n nginx

#View logs of the container
kubectl logs nginx -n nginx

#Get a shell inside the Pod
kubectl exec -it nginx -n nginx -- bash
```

Creating Deployment

Now that you've created your first Pod, you might wonder:
"Why do we even need a Deployment?"

Here's the simple answer:

- ✔ A Pod runs your container
- ✘ But a Pod cannot heal itself
- ✘ A Pod cannot scale itself
- ✘ A Pod cannot roll back or update itself

That's where Deployments come in. A Deployment makes your app:

Scalable (more Pods)
Self-healing (recreate Pods if they die)
Upgradable (rolling updates)
Rollback-able (undo failed updates)

This is why Deployments are the most commonly used controller in Kubernetes.

Different Types of Workload Controllers. Before writing YAML, let's briefly compare the main types:

Feature	ReplicaSet	Deployment	StatefulSet
Replicas	✅ Yes	✅ Yes	✅ Yes
Rolling updates	❌ No	✅ Yes	✅ Yes
Rollbacks	❌ No	✅ Yes	✅ Yes
Stable pod names	❌ No	❌ No	✅ Yes (ordered)
Persistent storage	❌ No	❌ No	✅ Yes

Typical use case	Internal use	Web apps, APIs	Databases, queues

Quick summary:

ReplicaSet = basic replication, no updates
Deployment = ReplicaSet + updates + rollbacks
StatefulSet = stable identity + storage (for DBs)

You DO NOT need to memorize different syntax — the YAML is mostly the same, just change the kind: field.

Here's the structure of a typical Deployment:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: <deployment-name>
  namespace: <namespace>
spec:
  replicas: <number>
  selector:
    matchLabels:
      app: <label>
  template:
    metadata:
      labels:
        app: <label>
    spec:
      containers:
      - name: <container-name>
        image: <image>
        ports:
        - containerPort: <port>
```

Explanation:
When you write a Deployment YAML, it always starts with three basic sections:

kind: tells Kubernetes *what* you are creating (a Deployment).
apiVersion: tells Kubernetes *which API version* to use.
metadata: gives your Deployment a *name* and *namespace.*

After that comes the main part:

spec: this is where you describe *what you want Kubernetes to do*. Inside spec, the first thing you provide is:

replicas: how many Pods you want.

selector: how the Deployment will identify its Pods (using labels).

template: this is your Pod configuration. Whatever you put inside template.spec is exactly what each Pod will look like.

This is why, When you create a Deployment, you do NOT need a separate pod.yml. The Deployment automatically creates Pods based on the template.

Actually yml for Deployment:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx-deployment
  namespace: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

What this means:

Creating 2 Pods of NGINX

Matching all Pods with app: nginx

Deployment controls Pod creation, scaling, and updates

Run command:

```
kubectl apply -f dep.yml
```

```
ubuntu@ip-172-31-31-163:~$ kubectl apply -f dep.yml
deployment.apps/nginx-deployment created
```

To verify

```
kubectl get deployment -n nginx
```

```
ubuntu@ip-172-31-31-163:~$ kubectl get deployment -n nginx
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   2/2     2            2           35s
```

To see the pods.

```
kubectl get pod -n nginx
```

```
ubuntu@ip-172-31-31-163:~$ kubectl get pod -n nginx
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-6b7f675859-xqk5h   1/1     Running   0          2m59s
nginx-deployment-6b7f675859-zqt5n   1/1     Running   0          2m59s
ubuntu@ip-172-31-31-163:~$
```

Scaling the Deployment

```
#syntax
kubectl scale deployment/<deployment-name> -n <namespace> --replicas=<number>

# actual command
kubectl scale deployment/nginx-deployment -n nginx --replicas=5
```

```
ubuntu@ip-172-31-31-163:~$ kubectl get pod -n nginx
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-6b7f675859-xqk5h   1/1     Running   0          9m16s
nginx-deployment-6b7f675859-zqt5n   1/1     Running   0          9m16s
ubuntu@ip-172-31-31-163:~$ kubectl scale deployment/nginx-deployment -n nginx --replicas=5
deployment.apps/nginx-deployment scaled
ubuntu@ip-172-31-31-163:~$ kubectl get pod -n nginx
NAME                                READY   STATUS              RESTARTS   AGE
nginx-deployment-6b7f675859-6qtd2   0/1     ContainerCreating   0          3s
nginx-deployment-6b7f675859-8g9df   0/1     ContainerCreating   0          3s
nginx-deployment-6b7f675859-99xpq   1/1     Running             0          3s
nginx-deployment-6b7f675859-xqk5h   1/1     Running             0          9m52s
nginx-deployment-6b7f675859-zqt5n   1/1     Running             0          9m52s
ubuntu@ip-172-31-31-163:~$
```

```
# Delete Deployment
kubectl delete -f dep.yml
```

```
# Diagnostic Commands (Very Useful)
# Describe the deployment
kubectl describe deployment nginx-deployment -n nginx
# Check rollout status
kubectl rollout status deployment/nginx-deployment -n nginx
# Undo the last update (rollback)
kubectl rollout undo deployment/nginx-deployment -n nginx
# View all ReplicaSets managed by this deployment
kubectl get rs -n nginx
```

Exercises

1. Create a ReplicaSet YAML with 3 replicas of nginx
2. Create a StatefulSet YAML with 2 replicas and stable names
3. Scale your Deployment to 6 Pods, then scale it back to 2
4. Update the Deployment to a new NGINX version
5. Roll back the Deployment to the previous version

Exposing Your Application (service)

Now that your Deployment is running NGINX Pods, there's one big question: How do users access your application?
You already have Pods running, but Pods:

- get new IPs every time they restart
- scale up and down
- move across nodes
- are not stable endpoints for users

This is why Kubernetes gives us Services. A Service provides a stable, permanent way to access your Pods. Even if:

- Pods die
- New Pods get created
- Pods get new IPs
- Pods shift to different nodes

Think of a Service as: A fixed door that always points to the correct Pod(s). Without a Service, no user outside the cluster, and not even other Pods inside the cluster, could reliably reach your application. A Service works using labels.

Your Deployment labels Pods like this:

```
app: nginx
```

Your Service "selects" Pods using the same label:

```
selector:
 app: nginx
```

Kubernetes automatically finds ALL Pods with that label and load-balances traffic between them.
Simple flow

```
User → Service → Pod → Container
```

No matter where Pods live (worker1, worker2, worker3), the Service handles the routing internally.

Types of Services

Kubernetes gives us 4 main types:

ClusterIP (default)
Internal-only access.
Useful when:
services inside the cluster need to talk to each other.
Example:
backend talking to database
Pods can access the Service, but the outside world cannot.

NodePort
Exposes the Service on a port of every node in the cluster. Kubernetes opens a port between 30000–32767 on each node. Users can access the app using:

```
<NodeIP>:<NodePort>
```

Good for:
local clusters
basic access
debugging

LoadBalancer
Used in cloud providers (AWS, GCP, Azure). Automatically creates a real cloud Load Balancer and exposes your app to the internet.
Example:
ELB on AWS
GLB on GCP
ALB/NLB on Azure
Most production apps use this type.

ExternalName
Maps a Service to an external DNS name. Useful when connecting to external services like:

```
mysql.company.com
api.paypal.com
```

No ports, no selectors. Just DNS mapping.

Creating a Service Using YAML

Here is your NGINX Service:

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
  namespace: nginx
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

Meaning:

Service name: nginx-service
It selects all Pods with app: nginx
Exposes port 80 inside the cluster
Sends traffic to Pod container port 80
Type is ClusterIP (internal-only)

Apply it:

```
kubectl apply -f service.yml
```

Check it:

```
kubectl get service -n nginx
```

```
ubuntu@ip-172-31-26-53:~$ kubectl get service -n nginx
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
nginx-service   ClusterIP   10.96.206.31   <none>        80/TCP    72m
ubuntu@ip-172-31-26-53:~$
```

As you can see, the ClusterIP of the Service is something like **10.96.206.31**, and the Service is listening on port **80**. But if you try to open that IP in your browser, like:

http://10.96.x.x:80

you won't get anything. Why? Because a ClusterIP is internal only.

This IP is:

NOT your EC2 public IP
NOT reachable from the internet
NOT even reachable from your host machine

It belongs to Kubernetes' own virtual network. That's why typing `10.x.x.x` in the browser never works. This behavior is completely normal. ClusterIP is strictly inside the cluster.

Since we are using Docker + Kind, the cluster is running inside Docker containers, so to access the Service from outside, we need port forwarding.

command:

```
kubectl port-forward service/nginx-service -n nginx 8080:80
--address=0.0.0.0
```

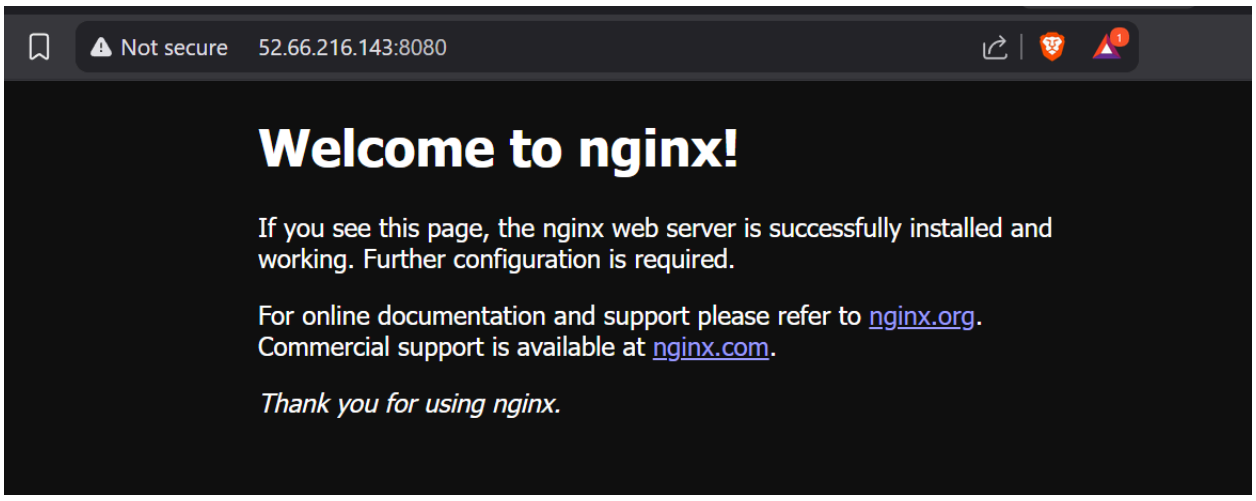
Port 81 is a privileged port (< 1024)
→ Linux blocks it unless you use sudo

But if you are still willing to use it. Use –E flag that will always be using your current user environment with the sudo privileges.

```
sudo -E kubectl port-forward service/nginx-service -n nginx 8080:80
--address=0.0.0.0
```

Now your application becomes accessible from your EC2 instance at:

http://<EC2-PUBLIC-IP>:8080



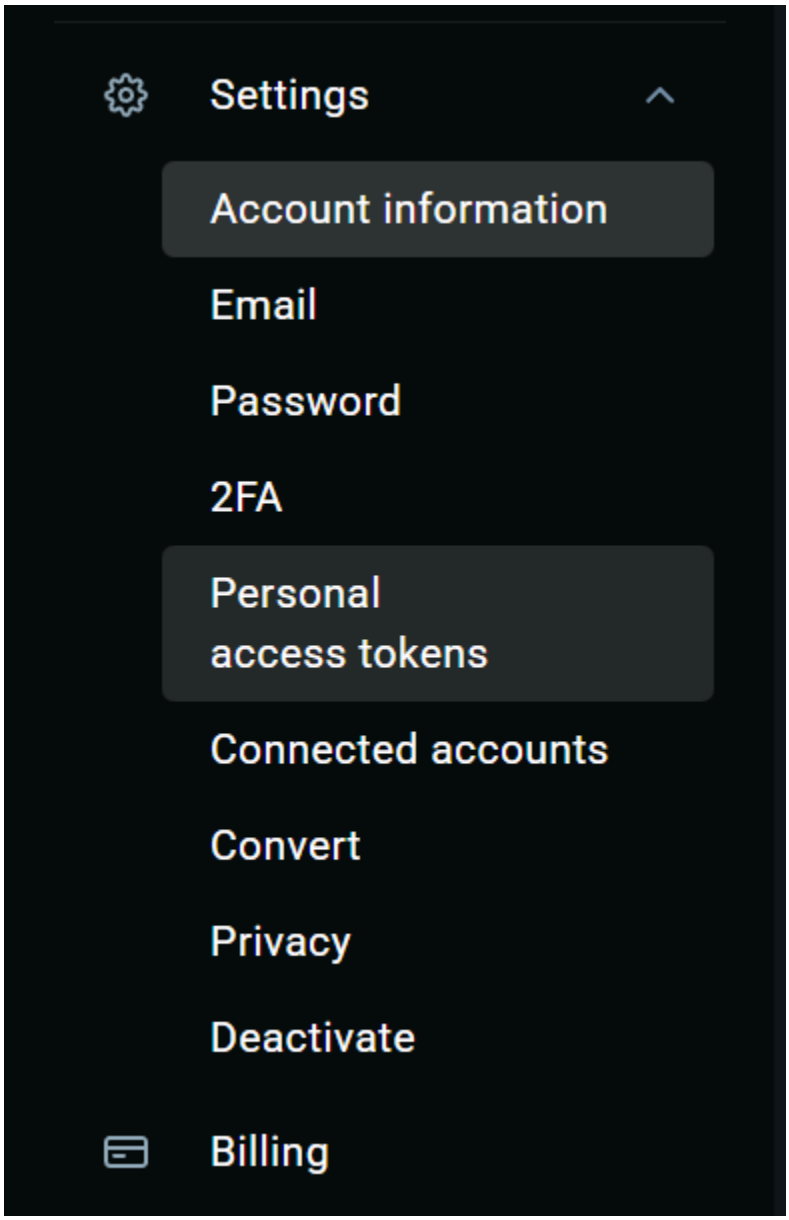
Deploy your first web-socket server on k8s

This exercise assumes you already understand Docker. I won't be covering Docker basics here.

Prerequisites:

- Solid Docker knowledge (multi-stage builds recommended)
- A Docker Hub account

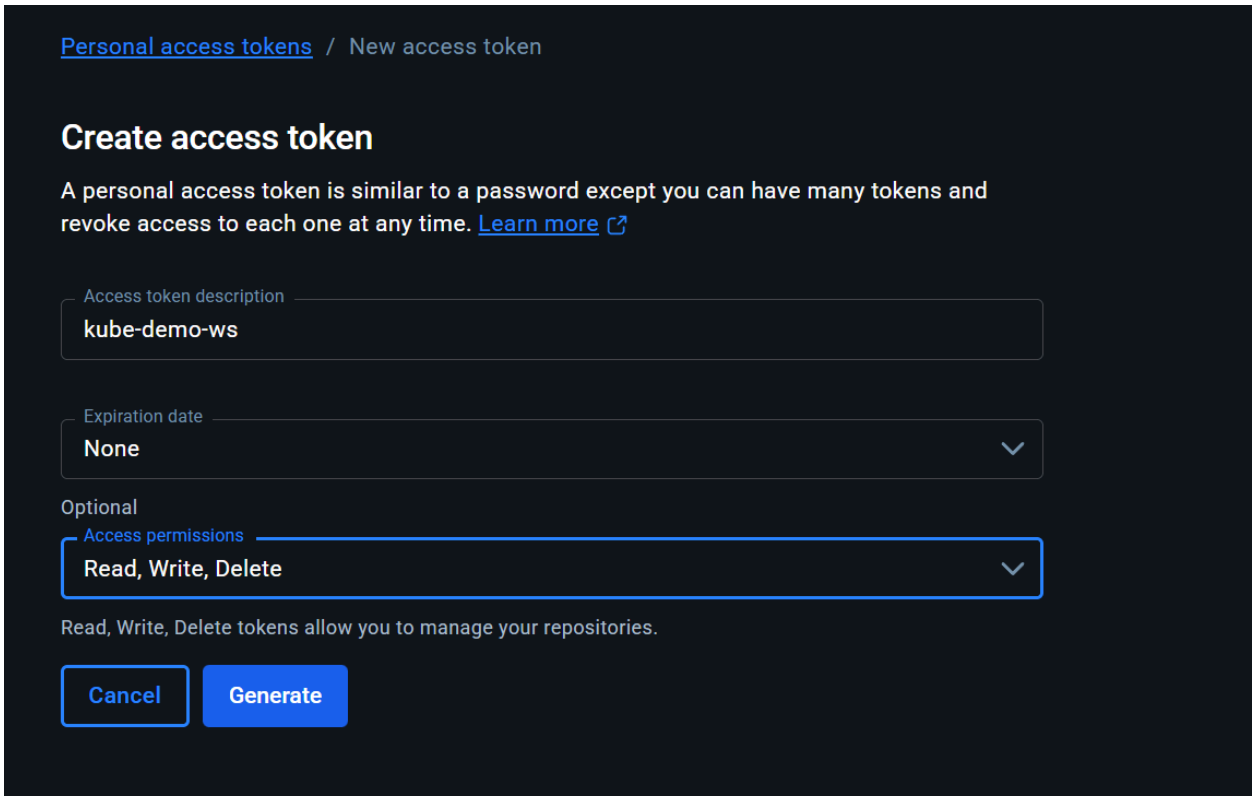
Go to your Docker Hub profile → Account Settings → Personal Access Tokens



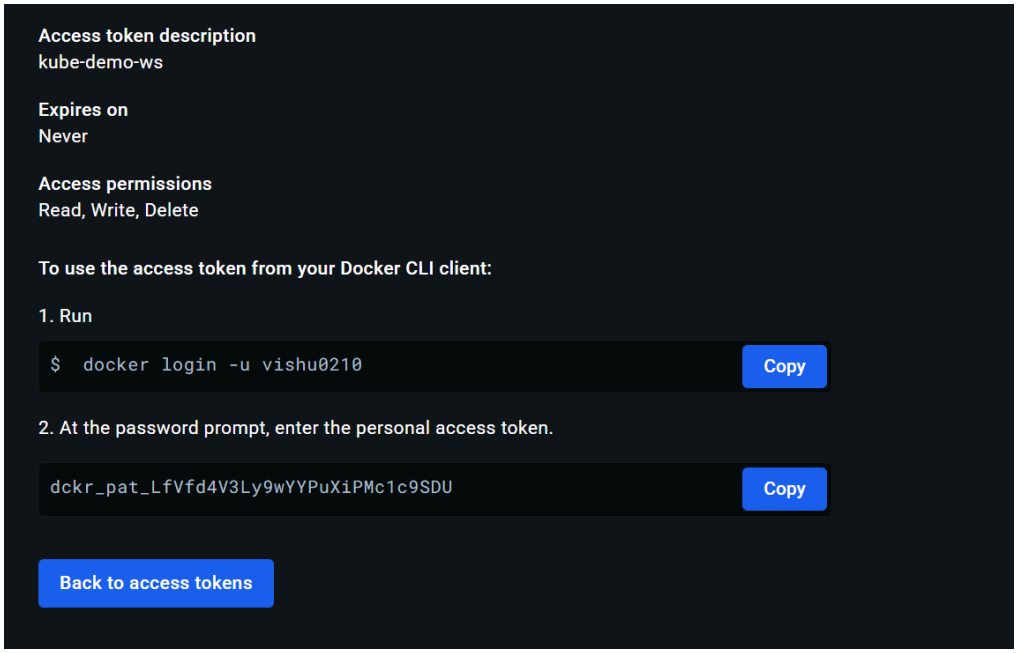
This personal access token allows our EC2 machine to authenticate with Docker Hub via the CLI.

We'll use it to push our Docker image from the EC2 instance and later pull it inside our Kubernetes setup through our development.yml.

Now generate a token on Docker Hub and give it the following permissions: read, write, delete.



And we are done here.



Now SSH into your EC2 instance (or your local machine) and create the cluster using this YAML file. This time, we're exposing port 8080 because our WebSocket server runs on that port. If you're using an EC2 instance, make sure to add port 8080 to your inbound security group rules as well.

Them kind-cluster.yml

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  image: kindest/node:v1.26.0
- role: worker
  image: kindest/node:v1.26.0
- role: worker
  image: kindest/node:v1.26.0
  extraPortMappings:
  - containerPort: 8080
    hostPort: 8080
    protocol: TCP
```

command:

```
kind create cluster -fonfig=kind-setup.yml
```

```
ubuntu@ip-172-31-18-77:~$ kind create cluster --config=kind-setup.yml
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.26.0) 🖼
 ✓ Preparing nodes 📦 📦 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
 ✓ Joining worker nodes 🚜
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Have a nice day! 👋
ubuntu@ip-172-31-18-77:~$
```

Now, let's get the app. For that we have to install git.

command:

```
sudo apt-get install git -y

git clone
https://github.com/vishutyagi0210/simple-ws-backend-Kubernetes-demo.g
it
```

```
ubuntu@ip-172-31-18-77:~$ ls
kind-setup.yml  kubectl.sha256  simple-ws-backend-Kubernetes-demo
ubuntu@ip-172-31-18-77:~$
```

Now, let's go inside and create the docker image.

```
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$ ls
Dockerfile  package-lock.json  package.json  src  tsconfig.json  tsconfig.tsbuildinfo
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$
```

Command:

```
docker build -t ws-backend .
```

```
found 0 vulnerabilities
npm notice
npm notice New major version of npm available! 10.8.2 -> 11.6.3
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.6.3
npm notice To update run: npm install -g npm@11.6.3
npm notice
 ---> Removed intermediate container 88fc15ed47f3
 ---> 3b276b379b61
Step 11/14 : COPY --from=builder /app/dist ./dist
 ---> 59962dcd59aa
Step 12/14 : RUN addgroup -g 1001 -S nodejs &&     adduser -S -u 1001 -G nodejs ws
 ---> Running in 30bc29b990c1
 ---> Removed intermediate container 30bc29b990c1
 ---> 2a7965f08dc8
Step 13/14 : RUN chown -R ws:nodejs /app
 ---> Running in 1dc47092ecb7
 ---> Removed intermediate container 1dc47092ecb7
 ---> 710a723b1da6
Step 14/14 : CMD ["node" , "dist/index.js"]
 ---> Running in ad8d346056ee
 ---> Removed intermediate container ad8d346056ee
 ---> 7ff37219f817
Successfully built 7ff37219f817
Successfully tagged ws-backend:latest
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$
```

Now push this image to Docker Hub. Use the credentials (personal access token) you created earlier — they will work here. Simply paste your Docker Hub username and token when the CLI prompts for login.

```
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$ docker login -u vishu0210

i Info → A Personal Access Token (PAT) can be used instead.
         To create a PAT, visit https://app.docker.com/settings


Password:

WARNING! Your credentials are stored unencrypted in '/home/ubuntu/.docker/config.json'.
Configure a credential helper to remove this warning. See
https://docs.docker.com/go/credential-store/

Login Succeeded
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$
```

Now, run these commands:

```
docker image tag ws-backend:latest <your-username>/ws-backend:latest
```

```
# then push

docker push <your-username>/ws-backend:latest
```

```
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$ docker push vishu0210/ws-backend:latest
The push refers to repository [docker.io/vishu0210/ws-backend]
dfe524dbf262: Pushing [==================================================>]  177.7kB
71b94d733053: Pushing [==================================================>]  10.75kB
34eb9e59d1cf: Pushing [==================================================>]   7.68kB
a7ad84ef889f: Pushing [==================================================>]    214kB
6ff992282ef6: Pushing [==================================================>]  5.632kB
f8eee56ccb51: Waiting
82140d9a70a7: Waiting
f3b40b0cdb1c: Waiting
0b1f26057bd0: Waiting
08000c18d16d: Waiting
```

Now it's time to create the Deployment file for this image. Everything will be similar to the NGINX Deployment we created earlier just updating the image and port details.

But first let's create the namespace

Command:

```
kubectl create ns websocket
```

Here is the dep.yml file for ws server

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: dep-ws-backend
  namespace: websocket
spec:
  replicas: 2
  selector:
    matchLabels:
      app: ws-backend
  template:
    metadata:
      labels:
        app: ws-backend
    spec:
      containers:
      - name: ws-backend
```

```
        image: vishu0210/ws-backend:latest
        ports:
        - containerPort: 8080
```

Command:

```
kubectl apply -f dep.yml
```

```
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$ kubectl apply -f ws-dep.yml
deployment.apps/dep-ws-backend created
```

Creating the server.yml

ws-service.yml

```
kind: Service
apiVersion: v1
metadata:
  name: ws-service
  namespace: websocket
spec:
  selector:
    app: ws-backend
  ports:
  - port: 8080
    targetPort: 8080
    protocol: TCP
  type: ClusterIP
```

Command:

```
kubectl apply -f ws-service.yml
```

```
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$ kubectl apply -f ws-service.yml
service/ws-service created
ubuntu@ip-172-31-18-77:~/simple-ws-backend-Kubernetes-demo$
```

Now, we just need one final command for port forwarding so that your local machine can send requests to the service running inside the cluster.

We're using local port 8081 because port 8080 is already occupied by a Docker container.

Port-forwarding maps:

local 8081 → service 8080 (inside Kubernetes)

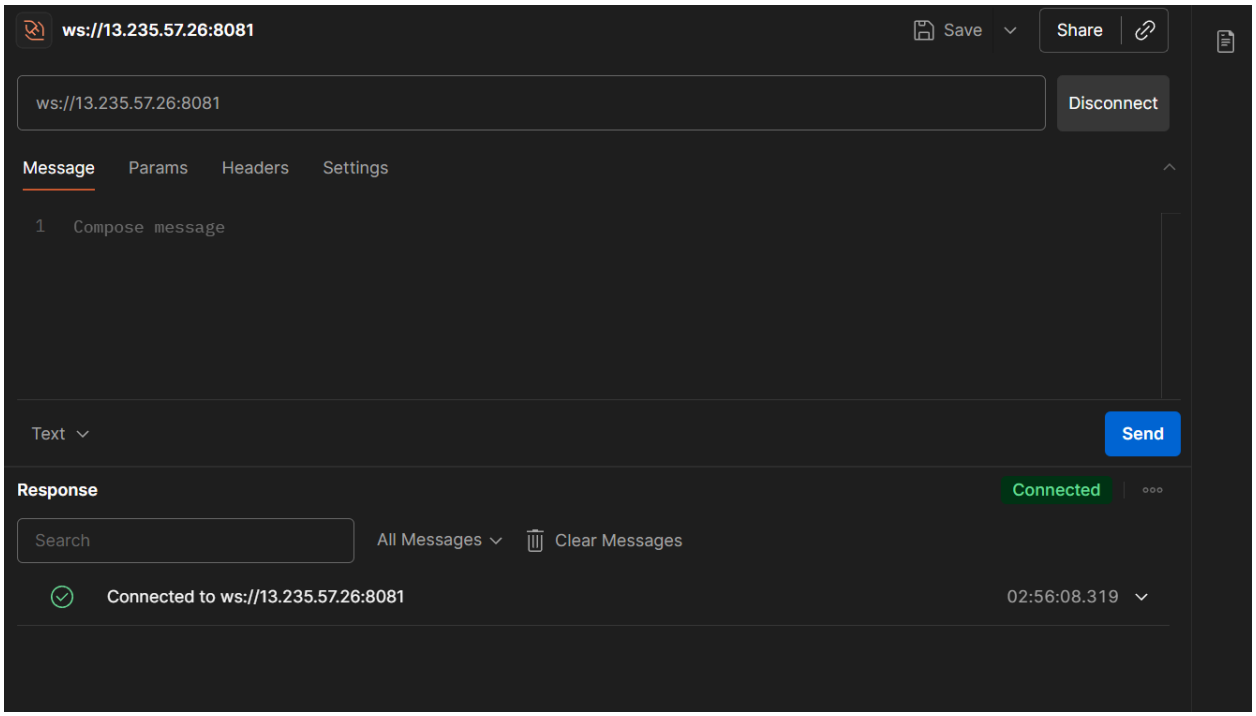
Command:

```
kubectl port-forward service/ws-service -n websocket 8081:8080
--address=0.0.0.0
```

Now open Postman (or any WebSocket client) and connect using:

Format.

ws://<instance-ip>:8081



Now, try sending and receiving some messages.

Payloads for the WebSocket server:

Join the room:

```
{
  "type": "join",
  "payload": {
    "roomId": "room123"
```

```
  }
}
```

Then send this payload:

Send message:

```
{
  "type": "send",
  "payload": {
    "roomId": "room123",
    "message": "Hello everyone!"
  }
}
```