



# Kubernetes

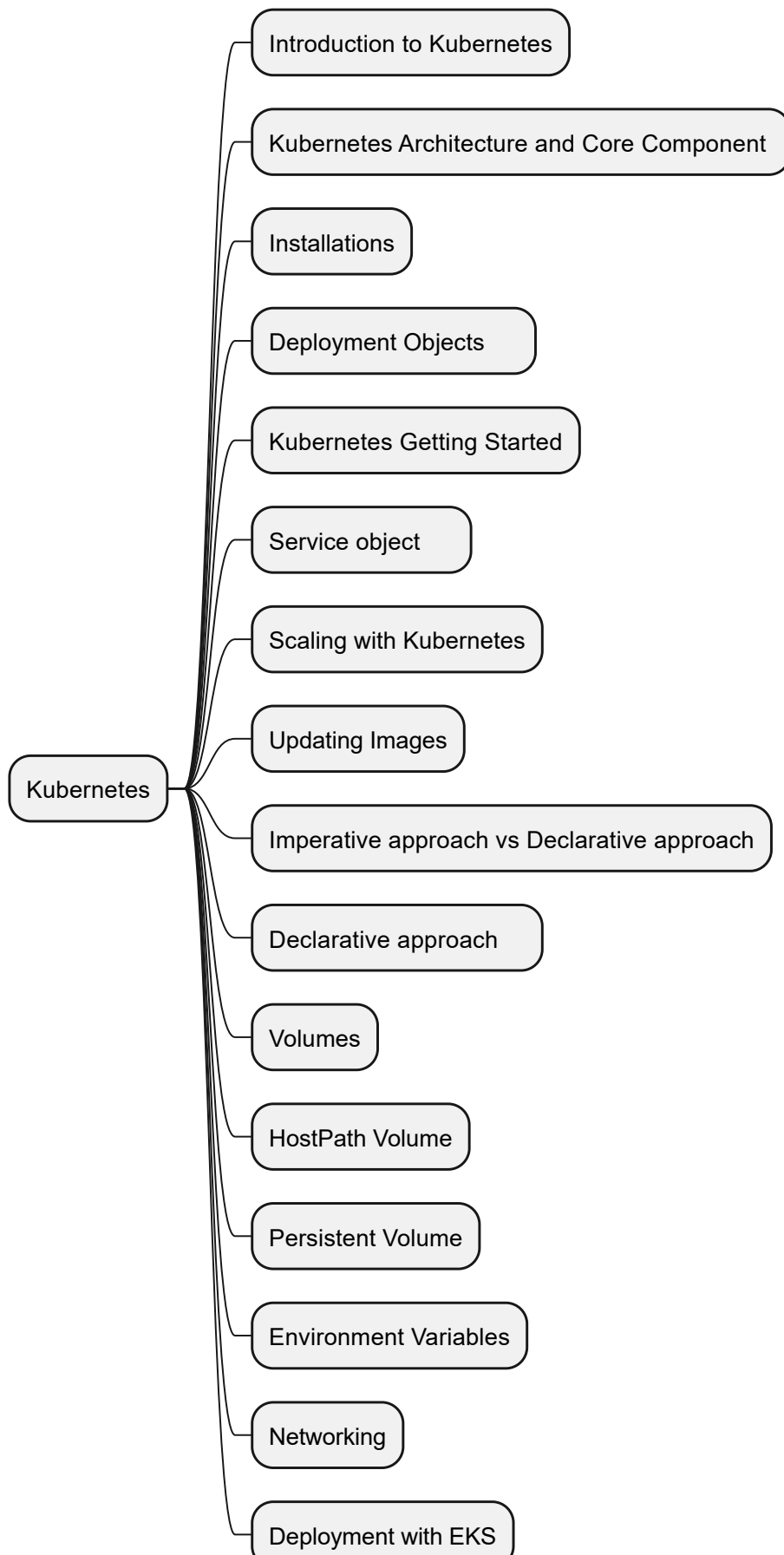
A Guide to Kubernetes

@Jonathan Ndambuki

# Table of contents

Kubernetes .....	2
Introduction to Kubernetes .....	5
Core Kubernetes Concepts & Architecture .....	8
Kubernetes Installation .....	12
Deployment Objects .....	14
Kubernetes Getting Started .....	20
Service Object .....	24
Scaling .....	27
Updating an Image .....	29
Imperative Approach VS Declarative approach .....	33
Declarative Approach .....	34
Volumes .....	41
Host Path Volume .....	48
Persistent Volume .....	50
Environment Variables .....	55
Kubernetes Networking .....	57
Deployment .....	66

# Kubernetes



\_\_\_\_\_

# Introduction to Kubernetes

## What is the Problem

Manual deployment of containers is hard to maintain , error-prone and annoying. Below are some of the challenges one might face:

- Containers might crash/ go down and need to be replaced
- We might need more container instances upon traffic spikes
- Incoming traffic should be distributed equally

## But an ECS seems to Solve the above issue

Yes, and Elastic Container Service(ECS) can solve the above issue but it locks us in, This might not sound as a major problem if you are using AWS but when you want to switch to another cloud provider its an issue because you will have to learn about the specifics, services and config options of the other cloud provider

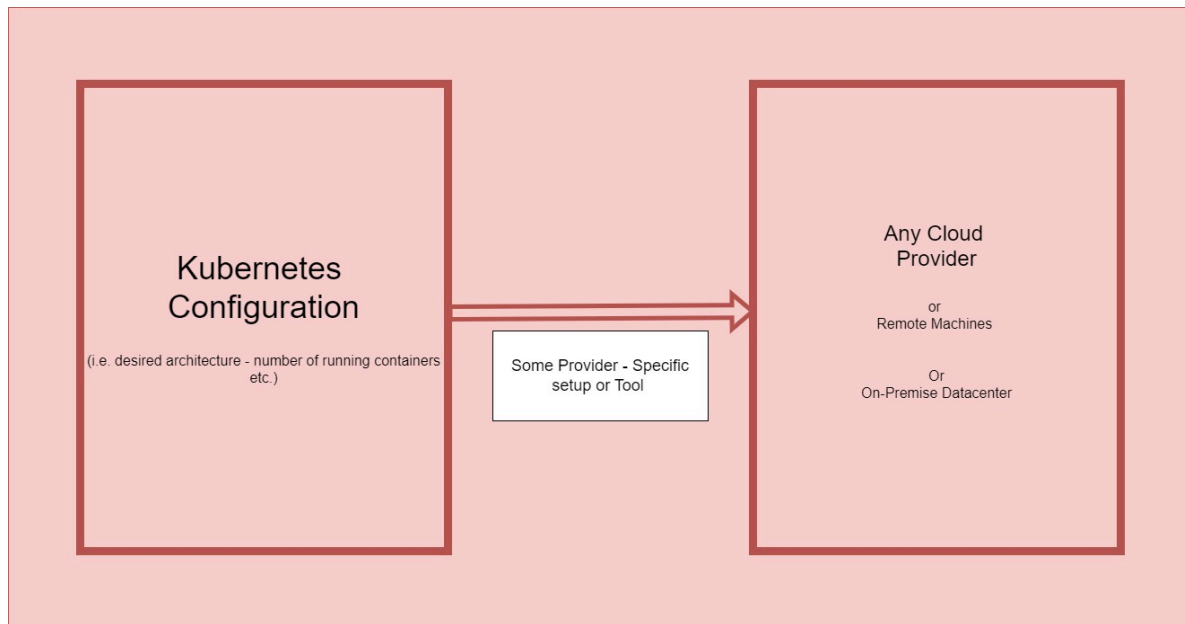
 **Kubernetes to the Rescue!**

## What is Kubernetes

Kubernetes, also known as K8s, is an open source system for automating deployment, scaling, and management of containerized applications

It provides a container runtime, container orchestration, container-centric infrastructure orchestration, self-healing mechanisms, service discovery and load balancing. It's used for the deployment, scaling, management, and composition of application containers across clusters of hosts.

Imagine having a bunch of servers, and you want to run your applications on them without worrying about where exactly they run or how they recover if something goes wrong. Kubernetes takes care of that for you, making sure your applications are always running smoothly.



k1.jpg

## What Kubernetes is not



- It's not a cloud service provider
- Its not a service by a cloud service provider
- It's not restricted to any specific (cloud) Service provider.
- Its not just a software you run on some machine
- its not an alternative to Docker
- its not a paid service

Rather it's:



- It is an open source project
- It can be used with any provider
- Its a collection of concepts and tools

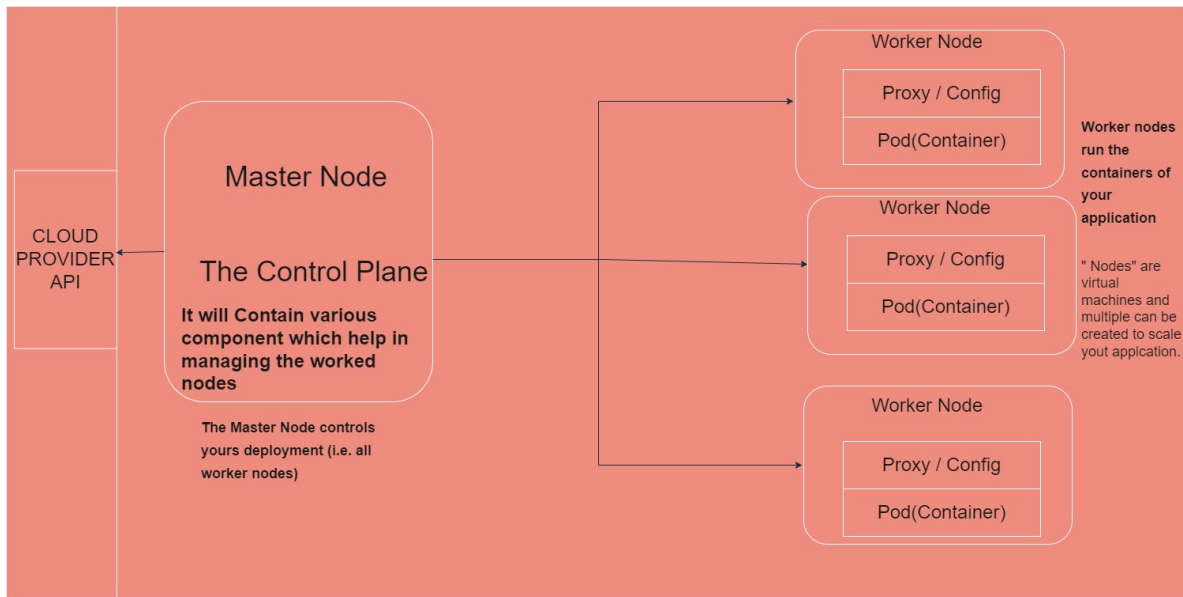
- It works with (docker) containers
- Its is a free open-source project
- its not a paid service



Kubernetes is like Docker-Compose for Multiple Machines



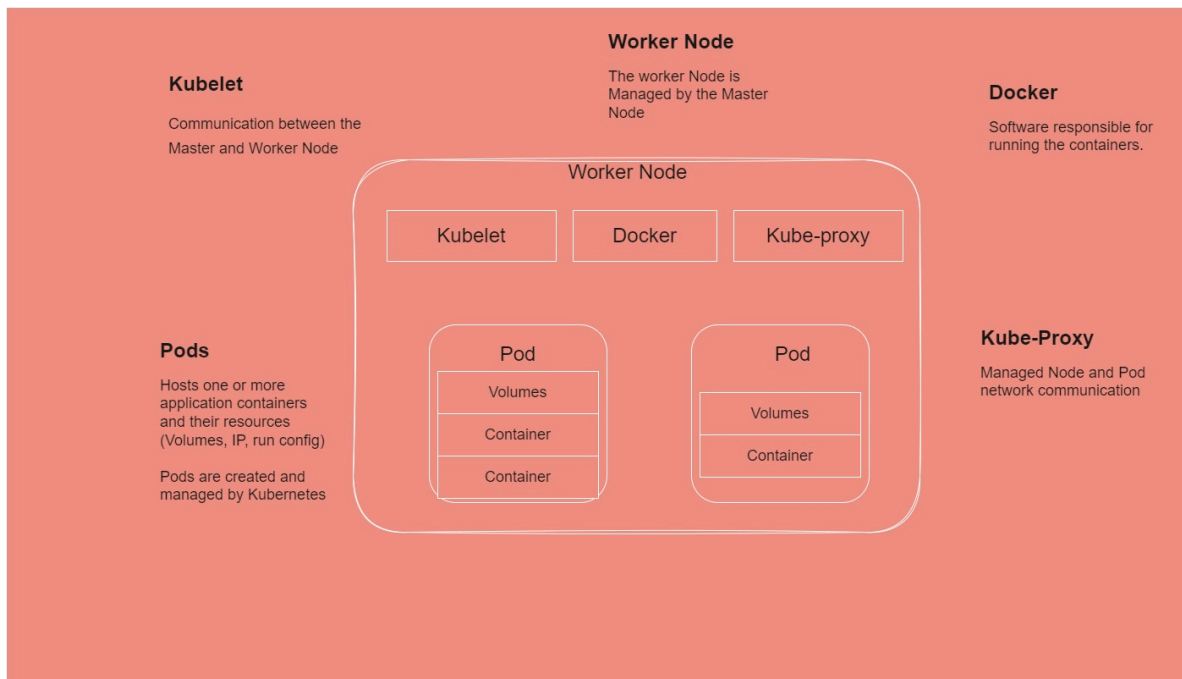
# Core Kubernetes Concepts & Architecture



k2.jpg

## The Worker Node

Think of the worker node as one computer/ machine / virtual instance These nodes run the container/containers and are managed by the master Node



k3.jpg

## Kubelet:

The Kubelet is an agent that runs on each node. It communicates with the Kubernetes control plane (the master node) to make sure the containers are running as expected. The Kubelet ensures that containers described in a PodSpec are running and healthy.

## Container Runtime:

This is the software responsible for running the containers. Common container runtimes include Docker, containerd, and CRI-O. The container runtime pulls the container images from a container registry and runs them as containers on the node. \

## Kube-proxy:

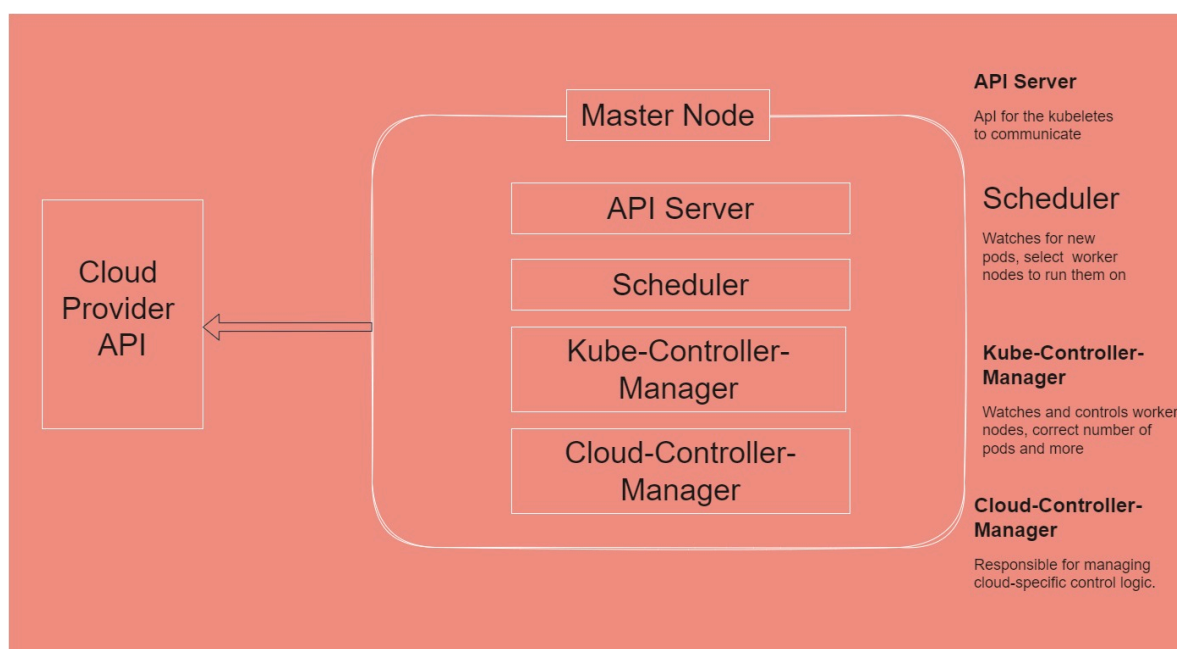
Kube-proxy is a network proxy that runs on each node in the Kubernetes cluster. It helps in managing the network rules and communication between services. It ensures that each service can reach any other service in the cluster, handling tasks like load balancing and routing traffic to the correct containers.

## Pods:

A Pod is the smallest, most basic deployable object in Kubernetes. Each pod represents a single instance of a running process in your cluster. Pods usually contain one or more containers that are tightly coupled and share resources like storage and networking.

## The Master Node

The master node (also known as the control plane) in Kubernetes is responsible for managing the entire cluster. It coordinates all activities within the cluster, like scheduling workloads, maintaining the desired state, and handling scaling and updates. Here's what's inside a Kubernetes master node:



k4.jpg

### API Server:

The API Server is the central management entity that exposes the Kubernetes API. It is the front-end for the Kubernetes control plane. All communication with the cluster, including interactions with the nodes, happens through the API Server. It handles RESTful requests and updates the state of the cluster accordingly.

### Controller Manager (kube-controller-manager):

The Controller Manager runs controllers, which are background processes responsible for maintaining the desired state of the cluster. Examples include the Node Controller (which handles node failures), Replication Controller (which ensures that the correct number of

pod replicas are running), and the Endpoints Controller (which manages endpoint objects tied to services).

## **Scheduler (kube-scheduler):**

The Scheduler is responsible for placing (or "scheduling") pods onto nodes in the cluster. It looks at the available resources on each node and matches that with the resource requirements of the pods. The scheduler makes decisions to ensure efficient resource utilization and adheres to any constraints defined in the pods' specifications.

## **Cloud Controller Manager (optional):**

The Cloud Controller Manager is responsible for managing cloud-specific control logic. It allows Kubernetes to interact with the underlying cloud provider, handling tasks like node provisioning, load balancers, and storage. This component is more relevant if you're running Kubernetes on a public cloud like AWS, GCP, or Azure.

# Kubernetes Installation

## We need to install Two Tools:

- Kubectl
- Minikube

## KubeCtl

The Kubernetes command-line tool, kubectl, allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs.

### Instaling KubeCtl

1. Download Chocolatey

Chocolatey (<https://chocolatey.org/install>)

2. Follow the steps in the link below to install Kubectl

Kubectl (<https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/#install-nonstandard-package-tools>)

## Installing Minikube

### Installing Minikube

- Installing Minikube

Minikube (<https://minikube.sigs.k8s.io/docs/start/?arch=%2Fwindows%2Fx86-64%2Fstable%2F.exe+download>)

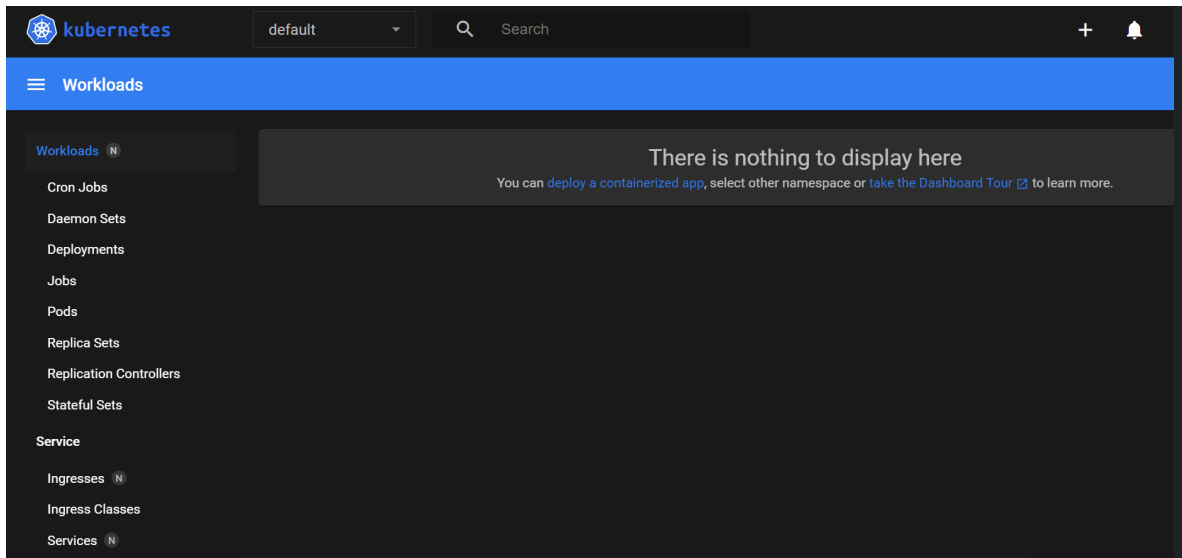
Now Test:

minikube is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes. All you need is Docker (or similarly compatible) container or a Virtual Machine environment

Run this On the Terminal:

```
minikube dashboard
```

The Output:



k3.PNG

# Deployment Objects

One of the key concepts in Kubernetes is the "desired state," which refers to the configurations of the applications that you want to deploy and run. Essentially, it's the way you want your applications to be set up and how you want them to behave in a Kubernetes environment. This includes things like how many instances of applications should be running, how those instances should be networked together, what resources they should have access to, and so on.

So, how do you define the desired state? By using objects.

## What Are Kubernetes Objects?

Objects are sort of like blueprints. They provide detailed instructions to Kubernetes on how the applications must be set up and managed. For example, a Deployment object might specify that you want three replicas (copies) of some application running at all times, while a Service object might define how you want to expose your application to the internet. Kubernetes then takes these instructions and automatically configures and manages the application accordingly, ensuring that it always matches the desired state.

### 1. Pods

Pods are the fundamental building blocks of the Kubernetes system. They are used to deploy, scale, and manage containerized applications in a cluster.

A Pod can host a single container or a group of containers that need to "sit closer together". By grouping two or more containers in a single Pod, they will be able to communicate much faster and share data more easily. It's important to note that, when a Pod contains multiple containers, all of the containers always run on a single worker node. They never span across multiple worker nodes. One important characteristic of Pods is that they are ephemeral in nature. This means that they are not guaranteed to have a long-term lifespan. They can be created, destroyed, and recreated at any time if required.

### 2. Deployment

In Kubernetes, a Deployment object is used to manage the lifecycle of one or more identical Pods. A Deployment allows you to declaratively manage the desired state of your application, such as the number of replicas, the image to use for the Pods, and the resources required. Declaratively managing the state means specifying the desired end

state of an application, rather than describing the steps to reach that state. Kubernetes figures out the steps on its own. So it will know what it has to do to launch the Deployment, according to the specifications provided by the user.

When you create a Deployment, you provide a Pod template, which defines the configuration of the Pods that the Deployment will manage. The Deployment then creates Pods that match this template, using a ReplicaSet. A ReplicaSet is responsible for creating and scaling Pods, and for ensuring that Pods that fail are replaced. When you update a Deployment, it will update the ReplicaSet, which in turn updates the Pods.

It's important to note that Deployment objects are used to manage stateless applications. Stateless applications are those that do not maintain any persistent data or state. This means that if a container running the application crashes or is terminated, it can be easily replaced. There's no need to preserve any data before deleting the old container and replacing it with a new one. Examples of stateless applications include web servers and load balancers.

### **3. ReplicaSets**

In Kubernetes, Deployments don't manage Pods directly. That's the job of the ReplicaSet object. When you create a Deployment in Kubernetes, a ReplicaSet is created automatically. The ReplicaSet ensures that the desired number of replicas (copies) are running at all times by creating or deleting Pods as needed.

To accomplish this, Kubernetes uses a concept called reconciliation loops. A reconciliation loop is a process that compares the desired state of a system with its current state and takes actions to bring the current state in line with the desired state.

Let's say you have specified that you want 3 replicas of a web server Pod to be running in your Kubernetes cluster. The reconciliation loop is constantly monitoring the current state of the cluster and comparing it to the desired state of 3 replicas. If it finds that there are only 2 replicas currently running, it will create a new replica to bring the current state in line with the desired state.

### **4. StatefulSet**

A StatefulSet is a Kubernetes object that is used to manage stateful applications. The word "state" means any stored data that the application or component needs to do its work. This state is typically stored in a persistent storage backend, such as a disk, or a database. And the state is maintained even if the underlying Pods/containers are recreated. The most



common example of a stateful component in applications is a database such as MySQL or MongoDB.

The StatefulSet ensures that each Pod is uniquely identified by a number, starting at zero. This allows for a consistent naming scheme for each Pod and its associated resources, such as persistent storage (for example, when Volumes are used).

When a Pod in a StatefulSet must be replaced, for example, due to node failure, the new Pod is assigned the same numeric identifier as the previous Pod. This ensures that the new Pod has the same unique identity and is attached to the same persistent storage (Volume) that the previous Pod was using.

This allows for the preservation of state and data across Pod replacements. This is important for applications that do work that periodically adds, removes, or changes some data. They basically need to remember "what has happened in the past" (previous state). For example, a Pod can add a new entry to a database, maybe the email address of a new user. If the Pod has to be recreated, you don't want to lose the email address that was previously added.

## 5. DaemonSets

A DaemonSet ensures that a copy of a Pod is running across all, or a subset of nodes in a Kubernetes cluster.

DaemonSets are useful for running system-level services, such as logging or monitoring agents, that need to run on every node in a cluster. Logging agents are used to collect log data from all the nodes in a cluster and send it to a centralized logging system for storage and analysis. Monitoring agents are used to collect metrics and performance data from all the nodes in a cluster, and send it to a monitoring system for analysis and alerting.

Like ReplicaSets, DaemonSets are managed by a reconciliation loop. A reconciliation loop is a mechanism that continuously checks and compares the desired state of a resource with the current state. The loop runs periodically and ensures that the DaemonSet is always in the desired state, automatically creating or deleting Pods as necessary.

## 6. PersistentVolume

PersistentVolume represents a piece of storage that you can attach to your Pod(s).

The reason it's called "persistent" is because it's not tied to the life cycle of your Pod. In other words, even if your Pod gets deleted, the PersistentVolume will survive.

And there are a lot of different types of storage that you can attach using a PersistentVolume, like local disks, network storage, and cloud storage.

There are a few different use cases for PersistentVolumes in Kubernetes. One common use case is for databases. If you're running a database inside a Pod, you'll likely want to store the database files on a separate piece of storage that can persist even if the Pod gets deleted. And PersistentVolume can do that.

## 7. Service

A Kubernetes Service is a way to access a group of Pods that provide the same functionality. It creates a single, consistent point of entry for clients to access the service, regardless of the location of the Pods.

For example, imagine you have a Kubernetes cluster with multiple Pods running a web application. Each Pod has its own IP address, but this can change at any time if the Pod is moved to another node, or recreated. So the IP address becomes a "moving target". The destination(s) that clients should reach is unstable, and hard to track.

To make it easier for clients to access the web application, you can create a Kubernetes Service that has a stable IP address. Clients can then connect to that IP, and their requests will be routed to one of the Pods running the web application.

One of the key benefits of using a Service is that it provides a stable endpoint that doesn't change even if the underlying Pods are recreated or replaced. This makes it much easier to update and maintain the application, as clients don't need to be updated with new IP addresses.

Furthermore, the Service also provides some simple load balancing. If clients would connect to a certain IP address of a specific Pod, that Pod would be overused, while the other ones would be sitting idle, doing nothing. But the Service can spread out requests to multiple Pods (load balance). By spreading these out, all Pods are used equally. However, each one has less work to do, as it only receives a small part of the total number of incoming requests.

## 8. Namespaces

A Kubernetes namespace is a way to divide a single Kubernetes cluster into multiple virtual clusters. This allows resources to be isolated from one another. Once a namespace is created, you can launch Kubernetes objects, like Pods, which will only exist in that namespace.

For example, imagine you have a Kubernetes cluster running two applications, "AppA" and "AppB". To keep things organized, you create two namespaces, "AppA-Namespace" and "AppB-Namespace".

Now, when you deploy Pods for "AppA", you can do so within the "AppA-Namespace". Similarly, when you deploy Pods for "AppB", you can do so within the "AppB-Namespace". What's a possible use case?

Well, imagine AppA, and AppB are almost identical. One is version 1.16 of an app, the other is version 1.17. They use almost the same objects, the same Pod structures, the same Services, and so on. Since they're so similar, there's a risk of them interfering with each other. For example, AppA might accidentally send requests to a similar Service or Pod used by AppB. But you want to test the new 1.17 version in a realistic scenario in the same cluster, using the same objects and definitions.

By using namespaces, you can perform as many operations as you need while eliminating the risk of impacting resources that are in another namespace. It's almost as if you have a second Kubernetes cluster. AppA runs in its own (virtual) cluster. AppB runs in a separate (virtual) cluster. But you don't actually have to go through the trouble of setting up an additional cluster. AppA and AppB are logically isolated from each other when they exist in separate namespaces. Even if they run identical Pods that want to access Services with identical names, there's no risk of them interfering with each other.

## 9-10. ConfigMaps & Secrets

ConfigMaps and Secrets are two very important objects that allow you to configure the apps that run in your Pods. Configuring apps refers to setting various parameters or options that control the behaviors of the apps. This can include things like database connection strings or API keys.

ConfigMaps are used to store non-sensitive configuration values. For example, environment variables used to provide runtime configuration information such as the URL of an external API, rather than confidential information such as passwords, are considered non-sensitive data.

Secrets, on the other hand, are meant to hold sensitive configuration values, such as database passwords, API keys, and other information that only authorized apps should be able to access.

ConfigMaps and Secrets can be injected into Pods with the help of environment variables, command-line arguments, or configuration files included in the volumes attached to those

Pods.

By using ConfigMaps and Secrets, you decouple the applications running in your Pods from their configuration values. This means you can easily update the configuration of your applications without having to rebuild or redeploy them.

## 11. Job

A job object is used to run specific tasks that have the following properties:

They are short-lived. They need to be executed once. But most importantly, Kubernetes has to make sure that the task has been executed correctly and finished its job. An example of where a Kubernetes job can be useful is a database backup. This does not run continuously, so it's a short-lived process. It just needs to start, complete, then exit. It has to be executed once. Even if the backup needs to happen weekly, there will be one separate job per week (CronJobs can be used for jobs that repeat periodically).

Each job has to finish its own weekly task. But it needs to ensure that the Pod executing the backup fully completes this job. For example, a Pod might start a backup. But the database is huge, so this can take hours. For some reason, the backup process fails at 68% progress. Kubernetes sees that the Pod has failed the task, so it can create another one to retry. It will keep on retrying until, finally, one Pod manages to back up the entire database.

In this case, the job required one Pod, and one successful run for one Pod (to ensure the task was completed successfully). But other jobs might require multiple Pods (which can run in parallel) and/or multiple successful runs. For example, a job will be considered complete, only when at least 3 Pods had successful runs and achieved 100% progress on their tasks.



But right Now this may not make sense So Lets dive into the practical Part

# Kubernetes Getting Started

Let's try running a project on Kubernetes

Below is the code Snippet:

```
const express= require('express')
const app = express()
app.get("/", (req,res)=>{
  res.send(`
    <h1> Hello Kubernetes </h1>
  `)
})
app.get('/error', (req,res)=>{
  process.exit(1)
})

app.listen(80, ()=>{
  console.log("App is Running...");
})
```

We Still need to Come up with a Docker Image (The image will be running in the pods)

Below are the Docker commands to create the Image:

```
FROM node:22-alpine

WORKDIR /app

COPY package*.json /app

RUN npm install

COPY . .

EXPOSE 80
```

```
CMD [ "npm", "start" ]
```

To build the Image Run

```
docker build -t node-app .
```

Let's create a Pod that will run this image, to do this we will use the deployment object, Run this Command:

```
kubectl create deployment node-object --image=node-app
```

This command will create a deployment object called node-object from an Image called node-app. The image is supposed to run in a pod which will be automatically created for you

To check the active Deployments Run:

```
kubectl get deployments
```

This Should be the output :

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
node-object	0/1	1	0	97s

image\_2.png

Now let's get all the Pods

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
node-object-6644cc887b-7trtx	0/1	ImagePullBackOff	0	8m21s

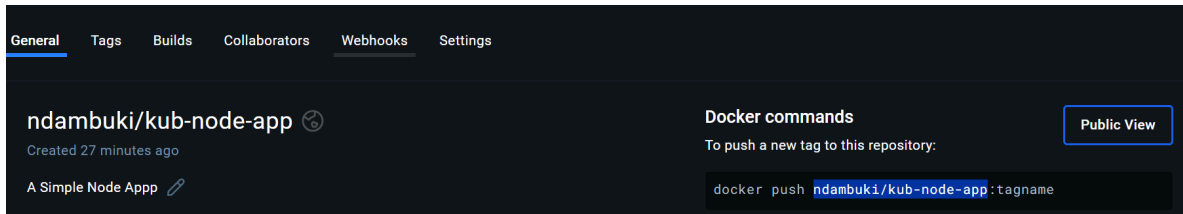
image\_3.png

The above might also display error in the status tab. This means that the image never built.

**⚠ The Cluster will try to pull the images from a Container Registry( e.g. DockerHub, ECR ) but in our case the Image is located locally.**

So now lets Push the Image to DockerHub, then the Cluster can pull from there.

Fist Create a repository in your DockerHub:



image\_5.png

Then tag you Image with the new Name

```
docker tag node-app ndambuki/kub-node-app
```

The Push the Image to your Docker Hub

```
docker push ndambuki/kub-node-app
```

Now we can Delete the Previous Deployment since it failed and Recreate again

Delete:

```
kubectl delete deployment node-object
```

Recreate and pass the right image( From DockerHub):

```
kubectl create deployment node-object --image=ndambuki/kub-node-app
```

Now when we get pods we should see the Container Running:

```
kubectl get pods
```

Output :

NAME	READY	STATUS	RESTARTS	AGE
node-object-db6bb7f9d-kdvd6	1/1	Running	0	46s

image\_6.png

But we still are not able to access the application : Lets look at another Object (Service object )



# Service Object

## Problem

We now want to access the Pod with the Node app, the pod has an IP address but this can change anytime if the pod is moved or recreated and this will make it hard to track. To make it easier for clients to access the web application, you can create a Kubernetes Service that has a stable IP address. Clients can then connect to that IP, and their requests will be routed to one of the Pods running the web application.

**One of the key benefits of using a Service is that it provides a stable endpoint that doesn't change even if the underlying Pods are recreated or replaced. This makes it much easier to update and maintain the application, as clients don't need to be updated with new IP addresses.**

Services will allow external access to pods.

```
kubectl expose deployment node-object --type=LoadBalancer --port=80
```

To Confirm Creation:

```
kubectl get services
```

Output:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	7d17h
node-object	LoadBalancer	10.105.184.48	<pending>	80:31999/TCP	35s

image\_7.png

Now to run the Service Run this on a terminal with admin privilege:

```
minikube service node-object
```

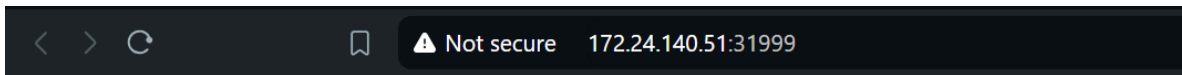
The above should run the container and the application on a browser or give you the URL to the site:

NAMESPACE	NAME	TARGET PORT	URL
default	node-object	80	http://172.24.140.51:31999

\* Opening service default/node-object in default browser...

image\_8.png

When you visit the Site you should see :



## Hello Kubernetes

image\_9.png

You can also access the Dashboard using:

```
minikube dashboard
```

This command will open the dashboard on a website, you can check your deployments and even pods :

Workloads > Pods							
Pods							
Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Us (bytes)
node-object-db6bb7f9d-kdvd6	ndambuki/kub-node-app	app: node-object pod-template-hash: db6bb7f9d	minikube	Running	0	-	-

image\_10.png

Now if you access the error route, which will crash the application. Kubernetes will automatically recreate a new pod but you will still be able to access the application using the same URI. **Thanks to services**

# Scaling

Scaling is one of the most important concept, because it focuses on availability of your application

## Scaling Up

In kubernetes to scale up and down we use a ReplicaSet. The ReplicaSet ensures that the desired number of replicas (copies) are running at all times by creating or deleting Pods as needed.

Below Example will create three replicas of the same Container:




```
kubectl scale deployment/node-object --replicas=3
```

Now If you check your Dashboard you will see two new pods:

Pods							
Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Us (bytes)
node-object-db6bb7f9d-8pbd4	ndambuki/kub-no-de-app	app: node-object pod-template-hash: db6bb7f9d	minikube	Running	0	-	-
node-object-db6bb7f9d-f626t	ndambuki/kub-no-de-app	app: node-object pod-template-hash: db6bb7f9d	minikube	Running	0	-	-
node-object-db6bb7f9d-kdvd6	ndambuki/kub-no-de-app	app: node-object pod-template-hash: db6bb7f9d	minikube	Running	2	-	-

image\_11.png

If you access the error route which will crash the container:

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Us (bytes)
 node-object-db6bb7f9d-8pbd4	ndambuki/kub-node-app	app: node-object pod-template-hash: db6bb7f9d	minikube	Running	0	-	-
 node-object-db6bb7f9d-f626t	ndambuki/kub-node-app	app: node-object pod-template-hash: db6bb7f9d	minikube	Error	0	-	-
 node-object-db6bb7f9d-kdvd6	ndambuki/kub-node-app	app: node-object pod-template-hash: db6bb7f9d	minikube	Running	2	-	-

image\_12.png

But Kubernetes will automatically replace the container with a new Healthy one. While one Container is down the other two pods can accept URL requests.

## Scale Down

Below Example will downscale from 3 pods to one :

```
kubectl scale deployment/node-object --replicas=1
```

If you check your dashboard you will have only one pod.

# Updating an Image

Sometimes you make changes to your code, rebuild the image and you want to see the changes appear.

Let's start with updating the code

```
const express= require('express')
const app = express()
app.get("/", (req,res)=>{
  res.send(`
    <h1> Hello Kubernetes </h1>
    <p> Some new Changes </p>
  `)
})
app.get('/error', (req,res)=>{
  process.exit(1)
})

app.listen(80, ()=>{
  console.log("App is Running...");
})
```

Now lets rebuild the image but for Kubernetes to Pick up the new Changes you must also tag the image with a different variation.

```
kubectl set image deployment/node-object kub-node-app=ndambuki/kub-node-app:V1
```

The set command will update an existing Image.You have to select the image you want to update and set it to the new image *kub-node-app=ndambuki/kub-node-app:V1*

You can always check the status of your update or rollout, in case of an Error it will be Highlighted.

An Example would be if we update with a image that does not exist:

```
kubectl set image deployment/node-object kub-node-app=ndambuki/kub-node-app:V3
```

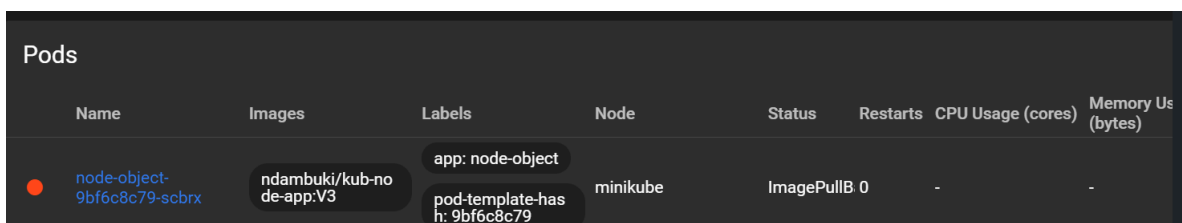
Now if you check the status of your Rollout using this command :

```
kubectl rollout status deployment/node-object
```

This will be the output:

```
Waiting for deployment "node-object" rollout to finish: 1 old replicas are pending termination...
```

and the Dashboard will also show an Error:



The screenshot shows the 'Pods' section of the Kubernetes Dashboard. It contains a table with columns: Name, Images, Labels, Node, Status, Restarts, CPU Usage (cores), and Memory Us (bytes). A single pod is listed with a red status icon. The pod name is 'node-object-9bf6c8c79-scbxr'. The image is 'ndambuki/kub-node-app:V3'. The labels are 'app: node-object' and 'pod-template-hash: 9bf6c8c79'. The node is 'minikube'. The status is 'ImagePullB. 0'. The restarts, CPU usage, and memory usage are all shown as '-'.

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Us (bytes)
node-object-9bf6c8c79-scbxr	ndambuki/kub-node-app:V3	app: node-object pod-template-hash: 9bf6c8c79	minikube	ImagePullB. 0	-	-	-

image\_14.png

To resolve the issue you can :

## Undo the Rollout

```
kubectl rollout undo deployment/first-app
```

or

## Set a Valid Image

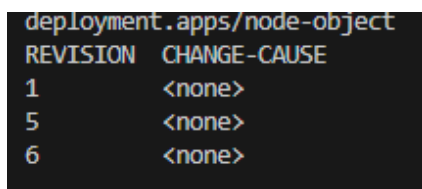
```
kubectl set image deployment/node-object kub-node-app=ndambuki/kub-node-app:V1
```

## Deployments History

To check the rollout history Run this command:

```
kubectl rollout history deployment/node-object
```

All Revision will be output:



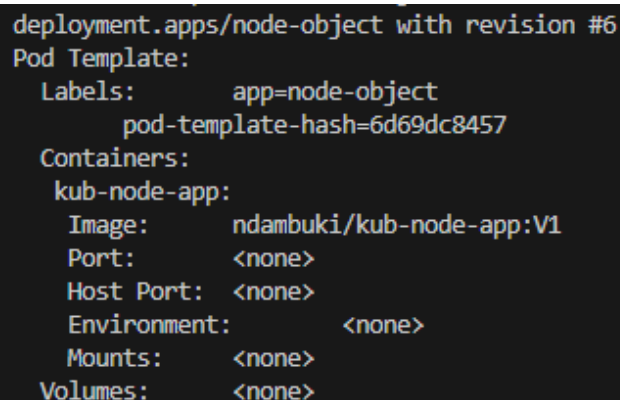
REVISION	CHANGE-CAUSE
1	<none>
5	<none>
6	<none>

image\_15.png

To show what a revision was about run this command:

```
kubectl rollout history deployment/node-object --revision=4
```

Output:



```
deployment.apps/node-object with revision #6
Pod Template:
  Labels:      app=node-object
              pod-template-hash=6d69dc8457
Containers:
  kub-node-app:
    Image:      ndambuki/kub-node-app:V1
    Port:       <none>
    Host Port:  <none>
    Environment: <none>
    Mounts:     <none>
    Volumes:    <none>
```

image\_16.png

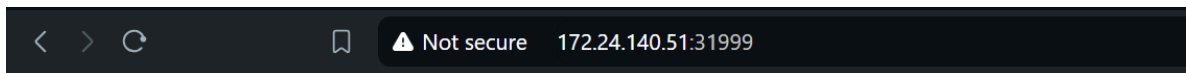
This will be able to show which image was used among other configurations.

You can Shift to a Previous revision using

```
kubectl rollout undo deployment/node-object --to-revision=1
```

Now this will show the first version that we had before updating





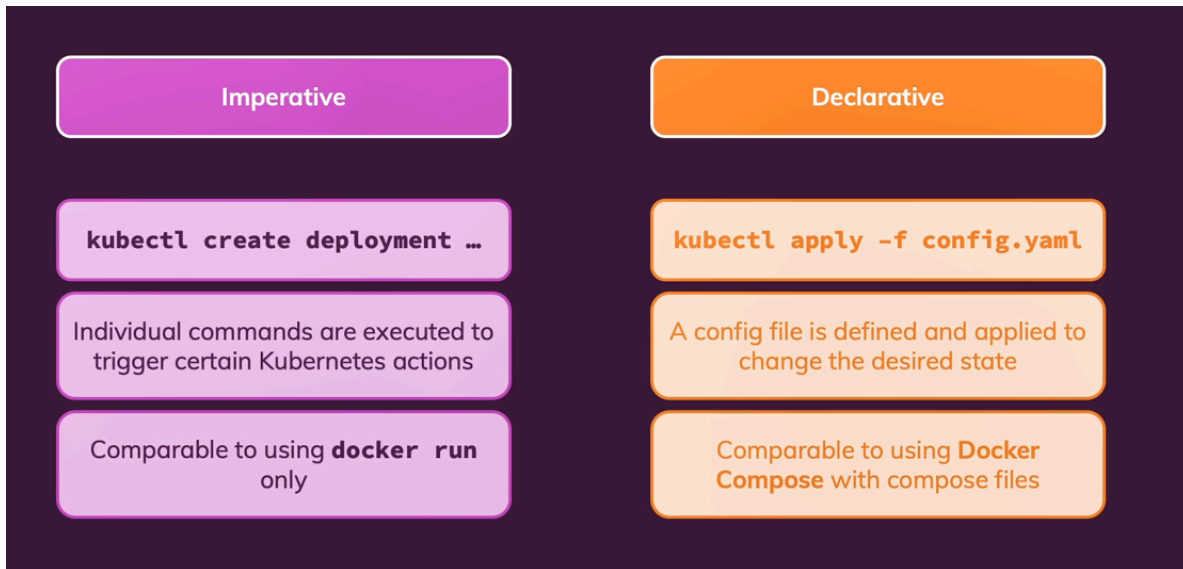
# Hello Kubernetes

image\_17.png

You can go back to the current version by running:

```
kubectl rollout undo deployment/node-object --to-revision=6
```

# Imperative Approach VS Declarative approach



image\_18.png

Its becoming tiresome to write commands on the terminal it's time we switch to the declarative approach where we can write the commands on a file.

# Declarative Approach

Having experienced the huddle of writing kubernetes commands on the terminal, I think its time we come up with a file Just like Docker has docker-compose Kubernetes can also receive instructions from a .yaml file

In your Project folder create a file and name it deployment.yaml (the name can vary but the extension must be .yaml)

Then copy these Commands to your \_yaml file , an explanation of the commands follows that.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: second-app-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: second-app
  template:
    metadata:
      labels:
        app: second-app
    spec:
      containers:
        - name: second-node-app
          image: ndambuki/kub-first-app:v1
```

## Deployment File Explanation:

### **apiVersion: apps/v1**

This specifies the version of kubernetes API you're using. In this case **apps/v1** is the API version used for managing applications like Deployments.

### **kind: Deployment**

The kind field tells Kubernetes what type of object you're creating. Here, it's a Deployment.

```
metadata:
  name: second-app-deployment
```

This section contains metadata for the deployment, like its name. This allows Kubernetes to uniquely identify this resource.

***name: second-app-deployment:*** This assigns a name to your Deployment, which is helpful for identifying and managing the Deployment later on

## Spec

The spec field defines the desired state for the deployment, including the number of replicas and the configuration of the pods.

### replicas: 1

This specifies the number of pod replicas you want to run. In this case, you want a single instance (replica) of the pod running.

### selector:

The selector tells the Deployment which pods it should manage. It selects pods that have the app: second-app label.

```
selector:
  matchLabels:
    app: second-app
```

***matchLabels: app: second-app:*** This ensures that only pods with the label app: second-app will be considered part of this deployment.

### template:

This defines the pod template that will be used for creating the pods

```
template:
  metadata:
    labels:
      app: second-app
```

metadata: labels: app: second-app: The pod template is labeled with app: second-app. This is crucial because the selector matches this label to know which pods the Deployment manages.

## containers:

This specifies the containers that will be created within the pods.

```
containers:
  - name: second-node-app
    image: ndambuki/kub-first-app:v1
```

- **name: second-node-app:** This is the name of the container.
- **image: ndambuki/kub-first-app:v1:** - This specifies the Docker image to be used by the container. Here, it's ndambuki/kub-first-app:v1, which likely refers to a Docker image hosted on a registry like Docker Hub. The tag v1 indicates the version of the image.



We have Created now the Deployment but we need a service to be able to access the application so lets create the service first the run both together

I a separate .yaml file copy the code below:

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: second-app
  ports:
    - protocol: 'TCP'
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

## Service File Explanation

## **apiVersion: v1**

This specifies the version of the Kubernetes API you're using. v1 is the version used for basic resources such as Service.

## **kind: Service**

The kind field tells Kubernetes what type of resource you're creating. Here, it's a Service, which is used to expose your application (usually a set of Pods) to the network.

## **metadata:**

This section contains metadata for the service, like its name. This is important for identifying and managing the Service within the Kubernetes cluster.

```
metadata:
  name: backend
```

**name: backend:** This assigns a name to your service, in this case, it's named backend.

## **spec**

The spec field defines the desired state for the service, including the configuration that tells the service how it should behave and how it should route traffic.

## **selector:**

The selector field tells the service which pods it should route traffic to, based on pod labels.

```
selector:
  app: second-app
```

**app: second-app:** This selector matches the pods with the label app: second-app, which corresponds to the label we defined in the deployment earlier. The service will route traffic to those pods.

## **ports**

This section defines the ports that the service will expose and how they map to the containers inside the pods.

```
ports:
  - protocol: 'TCP'
```

```
port: 80
targetPort: 8080
```

### **protocol: 'TCP'**

This specifies the protocol used for communication. In this case, it's TCP, the most common protocol for network communication.

### **port: 80**

This is the port on which the service will be exposed externally (i.e., the port clients will connect to).

### **targetPort: 8080**

This is the port on the container where the application is running. So, the service forwards traffic from port 80 to port 8080 on the matching pods.

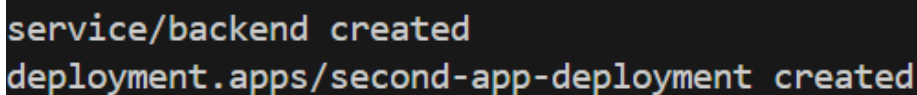
### **type: LoadBalancer**

The type field defines how the service is exposed externally. LoadBalancer: This means the service will be exposed through load balancer

Now We have the two Files lets run them

```
kubectl apply -f service.yaml -f deployment.yaml
```

You should receive these output:



```
service/backend created
deployment.apps/second-app-deployment created
```


image\_19.png

now start the service

```
minikube service backend
```

You should see the output In case you update something always rerun :

```
kubectl apply -f service.yaml -f deployment.yaml
```

 You can still have one .yaml file instead of two

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: second-app
  ports:
    - protocol: 'TCP'
      port: 80
      targetPort: 8080
  type: LoadBalancer

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: second-app-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: second-app
  template:
    metadata:
      labels:
        app: second-app
    spec:
      containers:
        - name: second-node-app
          image: ndambuki/kub-first-app:v1
```



Just make sure the different objects are separated by --- and also start with the service.

# Volumes

Attached is a Project find the Link here:

Volume Project (<https://github.com/joe6276/Kubernetes-101/tree/master/Kubernetes%20Projects/2.%20Kubernetes-Data>)

## Problem

The above project has a student folder which stores students that are added after executing the post endpoint. You can also get all the student by executing the get endpoint.

But when we run this in a pod and for one reason or another an error occurs (e.g. executing the error endpoint which crashes the app) , a new pod will be created and all the previous data will be lost.

This is what we will be solving: **persisting Data**

## EmptyDir

For a Pod that defines an emptyDir volume, the volume is created when the Pod is assigned to a node. As the name says, the emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

A container crashing does not remove a Pod from a node. The data in an emptyDir volume is safe across container crashes.

Here is the Dockerfile that will be built to create an image:

```
FROM node:22-alpine

WORKDIR /app

COPY package*.json /app

RUN npm install

COPY . .
```

```
EXPOSE 80
```

```
CMD [ "npm","start" ]
```

Let's start by creating a Service:

```
apiVersion: v1
kind: Service
metadata:
  name: volume-service
spec:
  selector:
    app: volumes
  ports:
    - protocol: 'TCP'
      port: 80
      targetPort: 80
  type: LoadBalancer
```

Then a Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: volume-object
spec:
  replicas: 1
  selector:
    matchLabels:
      app: volumes
  template:
    metadata:
      labels:
        app: volumes
    spec:
      containers:
        - name: volumes-container
```

```

        image: ndambuki/kub-volume-demo
        volumeMounts:
          - mountPath: /app/students
            name: student-volume
      volumes:
        - name: student-volume
          emptyDir: {}

```

Lets understand the Volume part:

```

spec:
  containers:
    - name: volumes-container
      image: ndambuki/kub-volume-demo
      volumeMounts:
        - mountPath: /app/students
          name: student-volume
  volumes:
    - name: student-volume
      emptyDir: {}

```

## Explanation

### volumeMounts:

Defines how the container will use volumes by mounting them to specific paths inside the container. **mountPath: /app/students:** This specifies the directory inside the container where the volume will be mounted. **name: student-volume:** Refers to the volume that will be mounted at /app/students. - the app id the work directory defined in the image.

### volumes:

Defines the volumes that will be available to the pod. These volumes can be of different types, such as emptyDir, hostPath, persistentVolumeClaim, etc. **name: student-volume:** The name of the volume that matches the volumeMount definition above.

**emptyDir: {}:** This creates an emptyDir volume, which is a temporary directory that initially starts empty. It exists as long as the pod is running, and when the pod is deleted, the data in the emptyDir is lost. It's typically used for temporary data storage during pod execution.

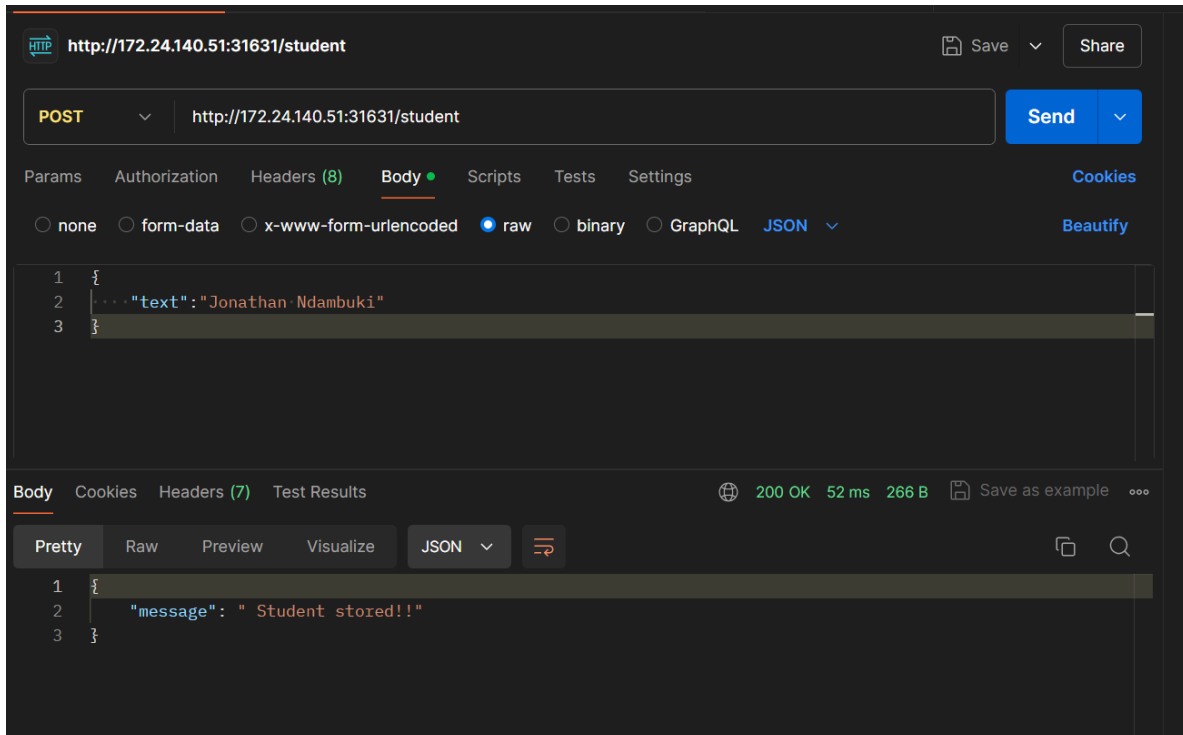
Now lets create the deployments:

```
kubectl apply -f service.yaml -f deployment.yaml
```

Start the service:

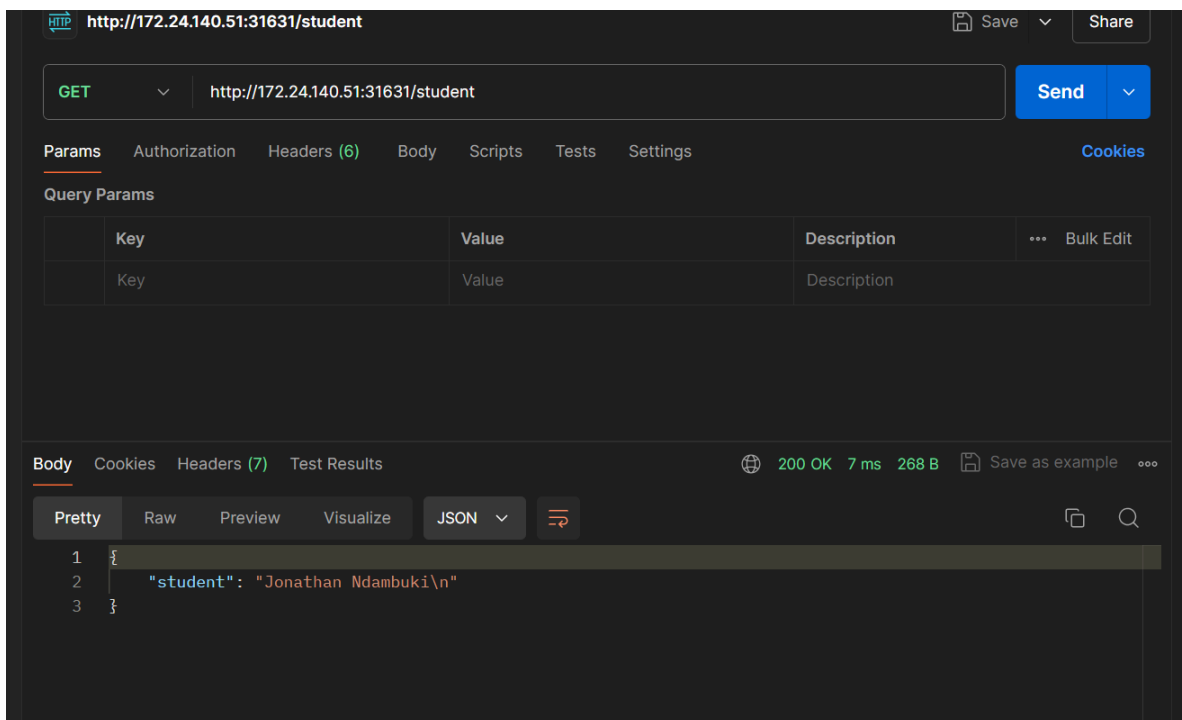
```
minikube service volume-service
```

Now lets test it using PostMan:



image\_20.png

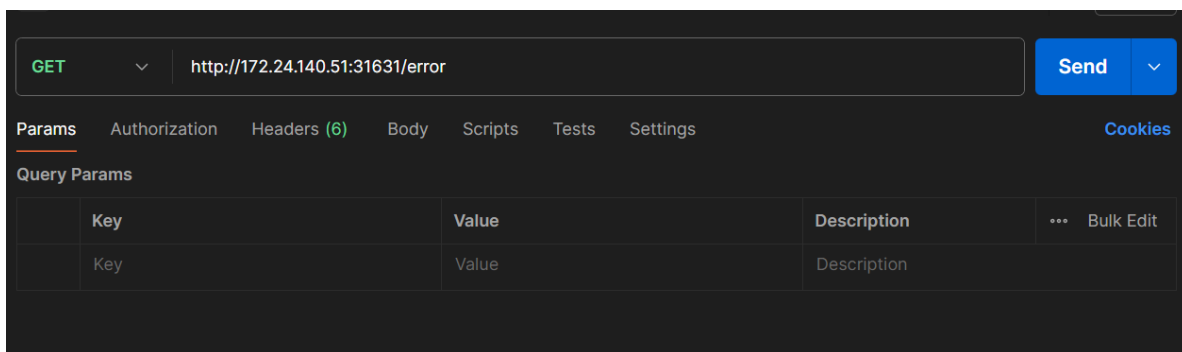
As you can see the student was stored Successful and we can confirm by executing the get endpoint.



image\_21.png

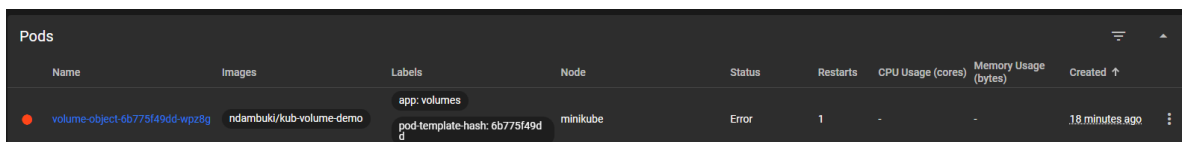
Now lets crash the app , which will force a new container to be created.

execute the error endpoint

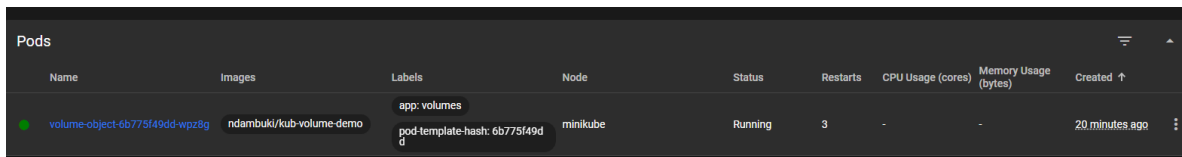


image\_23.png

on your dashboard you should see the container stopping and a new one will start.



image\_22.png



Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
volume-object-6b775f49dd-wp28g	ndambuki/kub-volume-demo	app: volumes pod-template-hash: 6b775f49d	minikube	Running	3	-	-	20 minutes ago

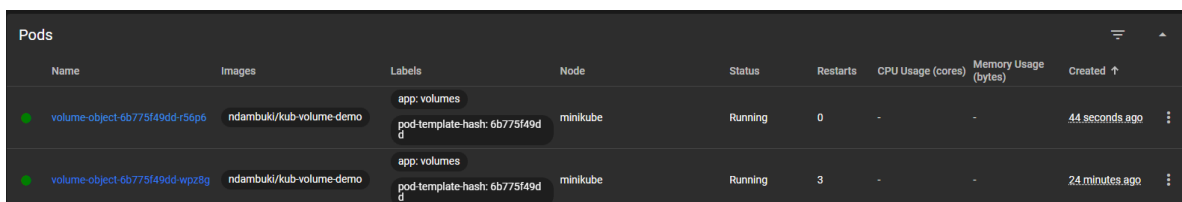
image\_24.png

**!** Our data still exists because EmptyDir will maintain the data despite container crashing

Now lets start two pods, modify the replica and re-apply the deployment.

```
spec:
  replicas: 2
```

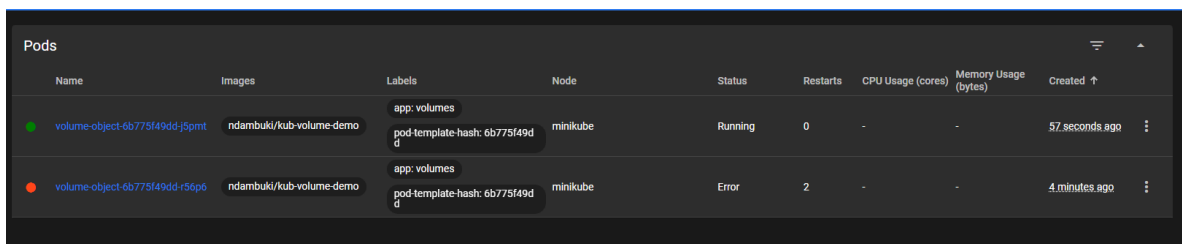
You should see the two pods in your dashboard:



Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
volume-object-6b775f49dd-r56p6	ndambuki/kub-volume-demo	app: volumes pod-template-hash: 6b775f49d	minikube	Running	0	-	-	44 seconds ago
volume-object-6b775f49dd-wp28g	ndambuki/kub-volume-demo	app: volumes pod-template-hash: 6b775f49d	minikube	Running	3	-	-	24 minutes ago

image\_25.png

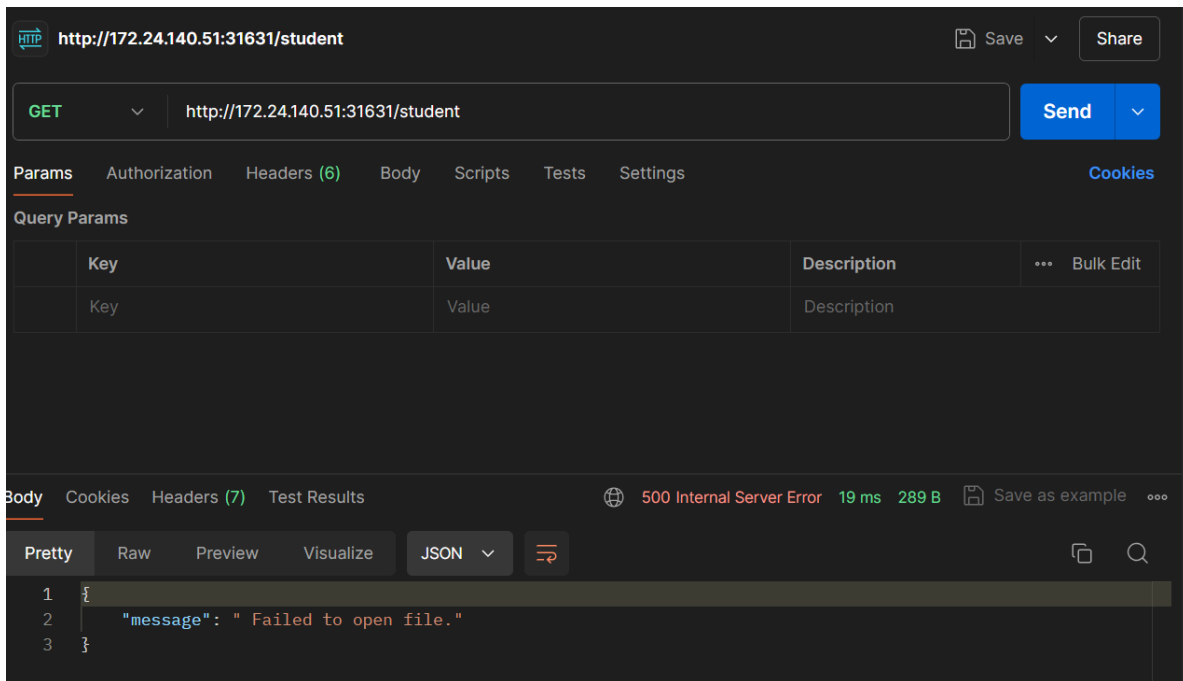
Now lets crash one which will force the new request to be forwarded to the active pod. Your dashboard should look like this (for some few seconds before the container is recreated)



Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
volume-object-6b775f49dd-j5pmt	ndambuki/kub-volume-demo	app: volumes pod-template-hash: 6b775f49d	minikube	Running	0	-	-	57 seconds ago
volume-object-6b775f49dd-r56p6	ndambuki/kub-volume-demo	app: volumes pod-template-hash: 6b775f49d	minikube	Error	2	-	-	4 minutes ago

image\_26.png

If you now try getting students the request, it will be forwarded to the new pod created and it will fail:



image\_27.png

⚠ EmptyDir is good when using one pod , but when you scale up to two pods it will fail.



# Host Path Volume

We saw that EmptyDir is pod dependent and we need a solution to that which is Host-path volume

## hostPath

A hostPath volume mounts a file or directory from the host node's filesystem into your Pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.

We are only going to change the volume configuration:

```
volumes:
  - name: student-volume
    hostPath:
      path: /data
      type: DirectoryOrCreate
```

### hostPath:

Specifies that this volume type is a hostPath, meaning it will use a directory or file from the host (the node running the pod) and mount it into the container.

### path: /data

This is the path on the host machine's filesystem that will be mounted into the container. In this case, it's /data. For example, if /data already exists on the host, it will be mounted into the container's filesystem at the corresponding mountPath defined in the container's configuration.


### type: DirectoryOrCreate


The type field describes how Kubernetes should handle the directory at the specified path.

**DirectoryOrCreate:** If the directory /data does not already exist on the host, Kubernetes will create it automatically. If the directory does exist, Kubernetes will use it as is. The directory will persist on the host node even after the pod is deleted.

now apply the changes

```
kubectl apply -f deployment.yaml
```

 This solves the previous problem with pods

 This volume however runs on one node only in a case where we are using different nodes (cloud) it might not be the best solution

# Persistent Volume

So far all the volumes we have interacted with are pod or node dependent. That means when you have new pods or nodes the volume won't work in the new pods or nodes. We need to create a volume at the cluster level, this will allow all pods and nodes in the cluster to access this volume. This is where persistent volume comes in.

Persistent Volumes are Kubernetes objects that represent storage resources in your cluster. PVs work in conjunction with Persistent Volume Claims (PVCs), another type of object which permits Pods to request access to PVs.

We will still use hostPath because we are using minikube and it has only one node.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: students-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data
    type: DirectoryOrCreate
```

## apiVersion: v1

Specifies the version of the Kubernetes API you're using. In this case, it's v1, used for resources like PersistentVolume.

## kind: PersistentVolume

The kind field indicates that you're creating a PersistentVolume (PV). A PV is a piece of storage in the cluster that has been provisioned by an administrator or dynamically by a storage class. Pods can claim PVs to store data persistently.

## metadata:

Contains metadata for the PersistentVolume. **name: host-pv:** This assigns a name to the PersistentVolume, in this case, host-pv. . **spec:** Defines the specification for the PersistentVolume, including the storage capacity, access modes, and volume source.

## capacity:

Specifies the size of the storage that the PersistentVolume provides.

```
capacity:  
  storage: 1Gi
```

**storage: 1Gi:** This indicates that the PersistentVolume has a storage capacity of 1 GiB .

## volumeMode: Filesystem

Defines the type of volume mode. **Filesystem:** This means that the volume will be mounted as a filesystem, where data can be written and read like any normal filesystem. The alternative would be **Block**, which gives raw block storage access.

## storageClassName: standard

Specifies the storage class associated with this PersistentVolume. **standard:** This binds the PersistentVolume to a specific StorageClass. In this case, it's using a standard storage class (which could be predefined or custom in your cluster). If a PersistentVolumeClaim (PVC) uses this storage class, it can claim this PV

## accessModes:

Specifies how the volume can be mounted by pods. **ReadWriteOnce:** This means the volume can be mounted as read-write by a single node at a time. Only one pod can write to this volume, but multiple pods can read from it if they are on the same node.

## hostPath:

Defines the backing storage for the PersistentVolume. Here, it uses the hostPath volume type **path: /data:** This specifies the directory on the host machine's filesystem that will back the PersistentVolume. In this case, it's /data.

**type: DirectoryOrCreate:** This ensures that if the /data directory doesn't exist on the host, it will be created automatically. If the directory exists, it will be used as is.

To connect to the volume we need PersistentVolumeClaim

## Persistent VolumeClaim

Persistent Volume Claims are requests for storage made by a pod in the cluster. They allow pods to consume storage without needing to know the specifics of how it is provisioned

Let's create one that consumes the above volume

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: students-pvc
spec:
  volumeName: students-pv
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:
    requests:
      storage: 1Gi
```

### apiVersion: v1

Specifies the version of the Kubernetes API being used. v1 is the appropriate API version for PersistentVolumeClaims.

### kind: PersistentVolumeClaim

The kind field indicates that you're creating a PersistentVolumeClaim (PVC). A PVC allows you to request storage from the cluster and bind it to a specific pod.

### metadata:

Contains metadata for the PersistentVolumeClaim. **name: host-pvc:** This gives the PersistentVolumeClaim a unique name, in this case, host-pvc

### spec:

Defines the specification of the PersistentVolumeClaim, including the access mode, storage request, and volume binding.

### **volumeName: host-pv**

This field explicitly binds the PersistentVolumeClaim to a specific PersistentVolume by name. **volumeName: host-pv:** The PVC will bind to the PersistentVolume named host-pv. This is useful when you have a statically provisioned PersistentVolume that you want to use directly with this claim

### **accessModes:**

Defines the access modes for the claim, specifying how the volume can be mounted.

**ReadWriteOnce:** This means the volume can be mounted as read-write by a single node at a time. The volume can be accessed by only one pod in read-write mode, but if the pod is terminated, another pod can then access it.

### **storageClassName: standard**

Specifies the StorageClass to be used for dynamic provisioning or matching against statically provisioned volumes.

**storageClassName: standard:** The PVC will look for a PersistentVolume that matches the standard storage class. This must match the storage class defined in the PersistentVolume.

### **resources:**

Specifies the resource requests, such as the amount of storage needed.

**requests: storage: 1Gi:** The PVC is requesting 1 GiB of storage. This must match the capacity of the PersistentVolume to be bound

In the Deployment file change this

```
volumes:
  - name: student-volume
    persistentVolumeClaim:
      claimName: students-pvc
```

This connects the pod to the persistent volume claim

Now Apply the changes

```
kubectl apply -f host-pv.yaml -f host-pvc.yaml -f deployment.yaml
```

The application should now run well .

# Environment Variables

Now let's check how to configure environment variables

First I will make the folderName to be read from the environment,

```
const filePath = path.join(__dirname, process.env.FOLDER_NAME, 'text.txt')
```

Now we have to pass a value for FOLDER\_NAME

```
spec:
  containers:
    - name: volumes-container
      image: ndambuki/kub-volume-demo
      env:
        - name: FOLDER_NAME
          value: 'students'
```

so we will pass key value pairs where the key will be FOLDER\_NAME and the value the name of the folder you want to use.

## ConfigMaps

### apiVersion: v1

Specifies the version of the Kubernetes API you're using. v1 is used for basic Kubernetes resources, such as ConfigMap.

### kind: ConfigMap

The kind field indicates that this resource is a ConfigMap. A ConfigMap is used to store configuration data that can be shared among different pods or services within your Kubernetes cluster.

### metadata:

Contains metadata, such as the name of the ConfigMap. **name: data-store-env:** This gives the ConfigMap a unique name, which can be referenced by pods or other resources that need the configuration.

### data:



The data section stores the key-value pairs of the configuration data. **folder: 'students':** This defines a key-value pair within the ConfigMap. The key is folder, and the value is 'students'.

This data can be injected into pods as environment variables or configuration files. For example, you could mount this ConfigMap into a pod, and the application inside the pod could read the value of the folder variable ('students').

```
- name: FOLDER_NAME
  valueFrom:
    configMapKeyRef:
      name: data-store-env
      key: folder
```

### **valueFrom:**

This indicates that the value of the environment variable will be sourced from an external resource, such as a ConfigMap or a Secret.

### **configMapKeyRef:**

This specifies that the value will come from a ConfigMap **name: data-store-env:** This specifies the name of the ConfigMap from which the value will be fetched. In this case, it is data-store-env. **key: folder:** This specifies the key inside the ConfigMap whose value you want to retrieve. In this case, it's the key folder, which has the value 'students'.

# Kubernetes Networking

When you have a full-stack application you need the frontend and the backend to communicate with each other, you also need the backend to communicate with your database.

For example if running an MSSQL the backend need a server name. In Docker you use the database container name as the server name. In kubernetes things are different lets check different scenarios :

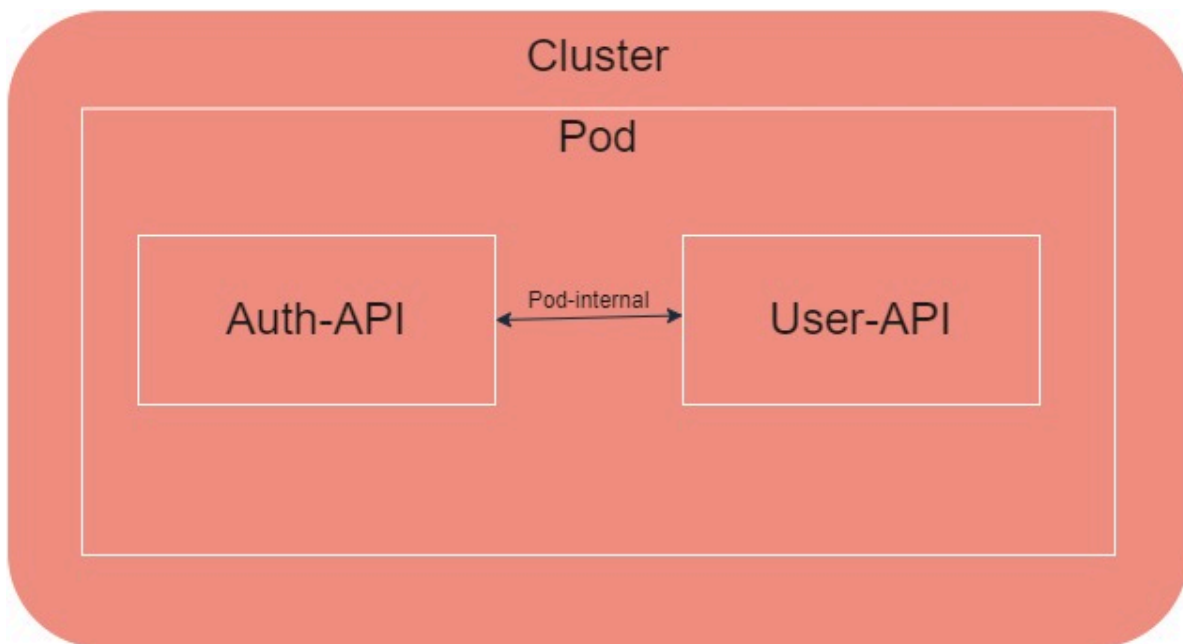
Attached Here is a Project: <https://github.com/joe6276/Kubernetes-101/tree/master/Kubernetes%20Projects/3.%20Kubernetes-%20Networking>

## About the Project

The project has two API auth and users . The users API needs to communicate to the Auth API in order to get back a hashed password( its simulated - not real) But let's focus on the communication part.

## Pod Internal Communication

We will start by placing both containers in the same pod. When two containers are in the same pod you can use localhost because they are running on the same machine. Here is an example of the deployment.



dsa.jpg

## user Service

```
apiVersion: v1
kind: Service
metadata:
  name: user-service
spec:
  selector:
    app: user
  ports:
    - protocol: 'TCP'
      port: 3000
      targetPort: 3000
  type: LoadBalancer
```

## User Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: user
  template:
    metadata:
      labels:
        app: user
    spec:
      containers:
        - name: users-api
          image: ndambuki/kubernetes-user-api:latest
          env:
            - name: AUTH_ADDRESS
              value: localhost
```

```
- name: users-auth
  image: ndambuki/kubernetes-auth-api:latest
```

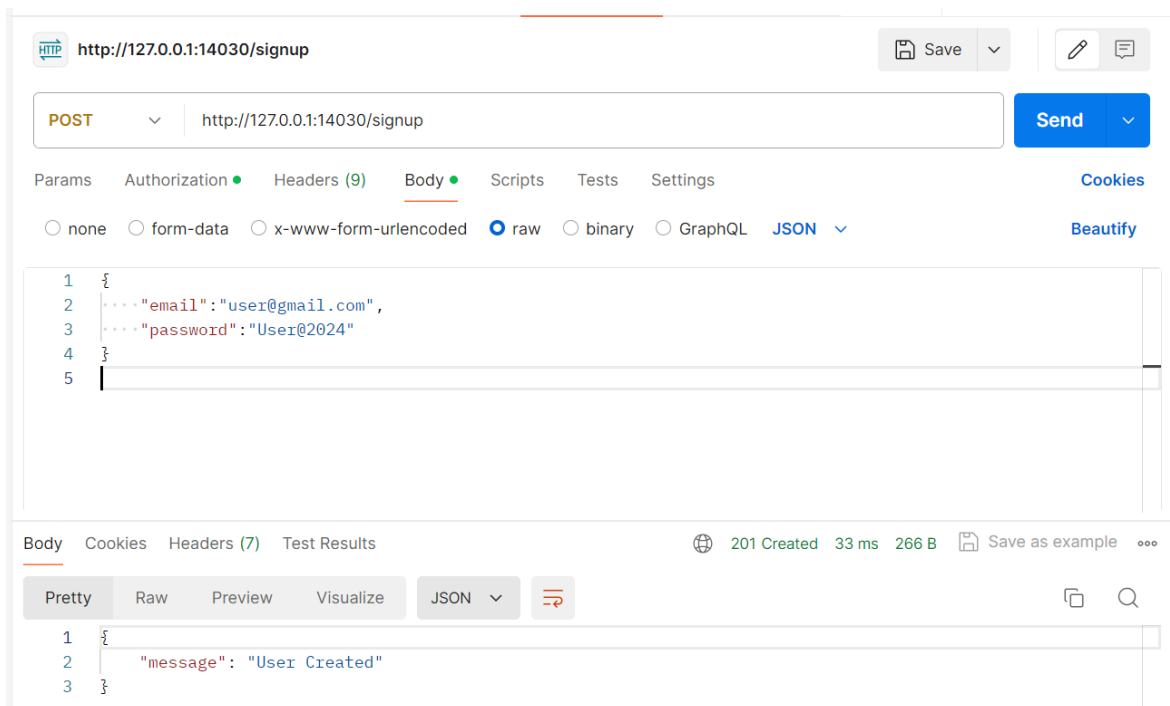
The deployment has two containers and I'm passing AUTH\_ADDRESS as an environment variable. which is being used at the User API

```
const response = await
axios.get(`http://${process.env.AUTH_ADDRESS}/hash/${password}`);
```

Now start the service with:

```
minikube service user-service
```

The Project Works Well

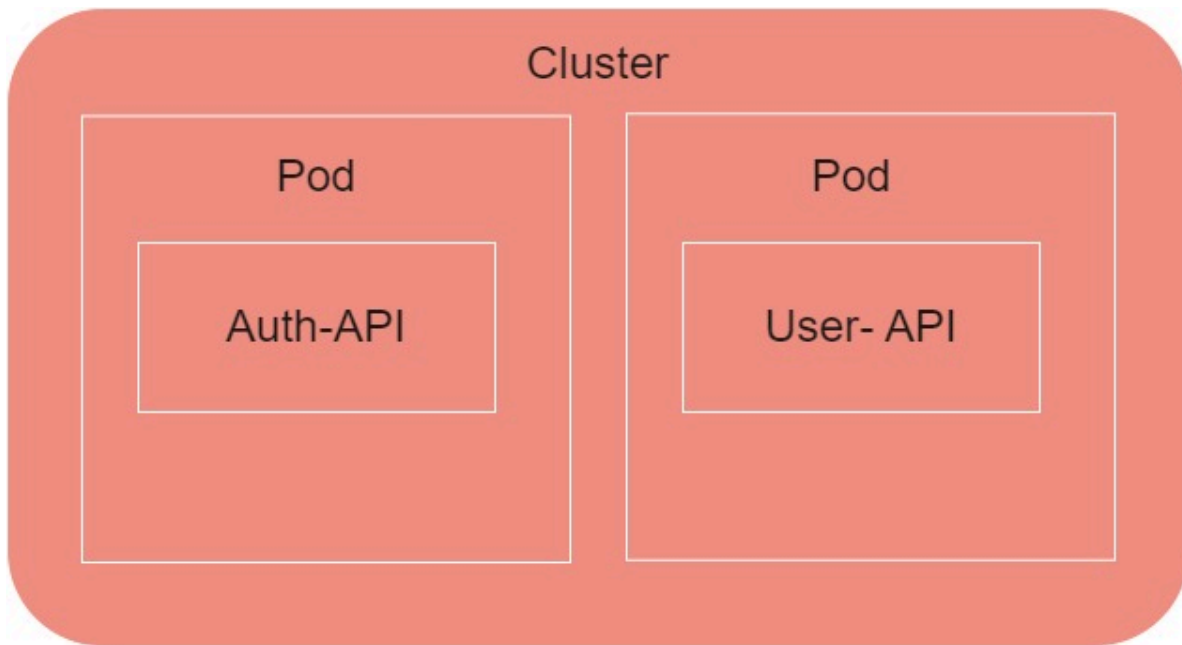


image\_52.png

But sometimes you have the containers on different pods.

## Pod to Pod Communication

### Auth-Service IP Address



dsa1.jpg

Now let's create two deployments and two services:

### Auth Service

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

Here I used ClusterIP which means that this service won't be accessed from the outside can only be accessed by pods in the same cluster.

### Auth Deployment

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth-api
          image: ndambuki/kubernetes-auth-api:latest

```

Before running the User Deployment lets first get the IP address of the AUTH Service. Run:

```
kubectl get services
```

:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
auth-service	ClusterIP	10.108.196.53	<none>	80/TCP	3m54s
fruits-service	LoadBalancer	10.102.233.249	<pending>	4000:30708/TCP	20h
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	20h
user-service	LoadBalancer	10.97.59.34	<pending>	3000:30318/TCP	65m

image\_53.png

Use the ClusterIP as the AUTH\_ADDRESS of the User deployment.

## User Deployment

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-deployment

```

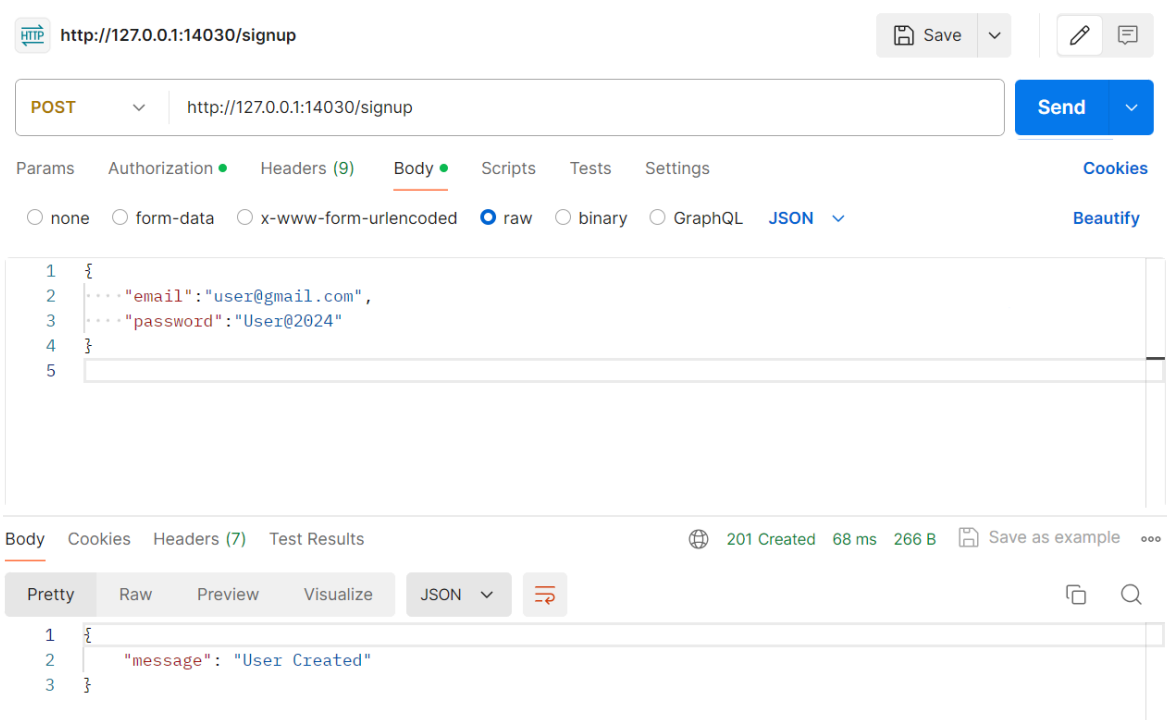
```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: user
  template:
    metadata:
      labels:
        app: user
    spec:
      containers:
        - name: users-api
          image: ndambuki/kubernetes-user-api:latest
          env:
            - name: AUTH_ADDRESS
              value: "10.108.196.53"
```

The User Service remains the same .

now apply all the changes

```
kubectl apply -f user.yaml -f user-service.yaml -f auth.yaml -f auth-
service.yaml
```

The application will still Work



image\_54.png

But that's a lot of manual work , in terms getting the IP addresses.

## DNS for Services and Pods

Kubernetes creates DNS records for Services and Pods. You can contact Services with consistent DNS names instead of IP addresses.

Kubernetes publishes information about Pods and Services which is used to program DNS. Kubelet configures Pods' DNS so that running containers can lookup Services by name rather than IP.

Services defined in the cluster are assigned DNS names. By default, a client Pod's DNS search list includes the Pod's own namespace and the cluster's default domain.

Cluster by default come with coreDNS which gives us cluster internal domain name for our services. Here you use: `<service_Name>.`

Namespace is a way of grouping your services. By default we use the 'default' namespace unless defined.

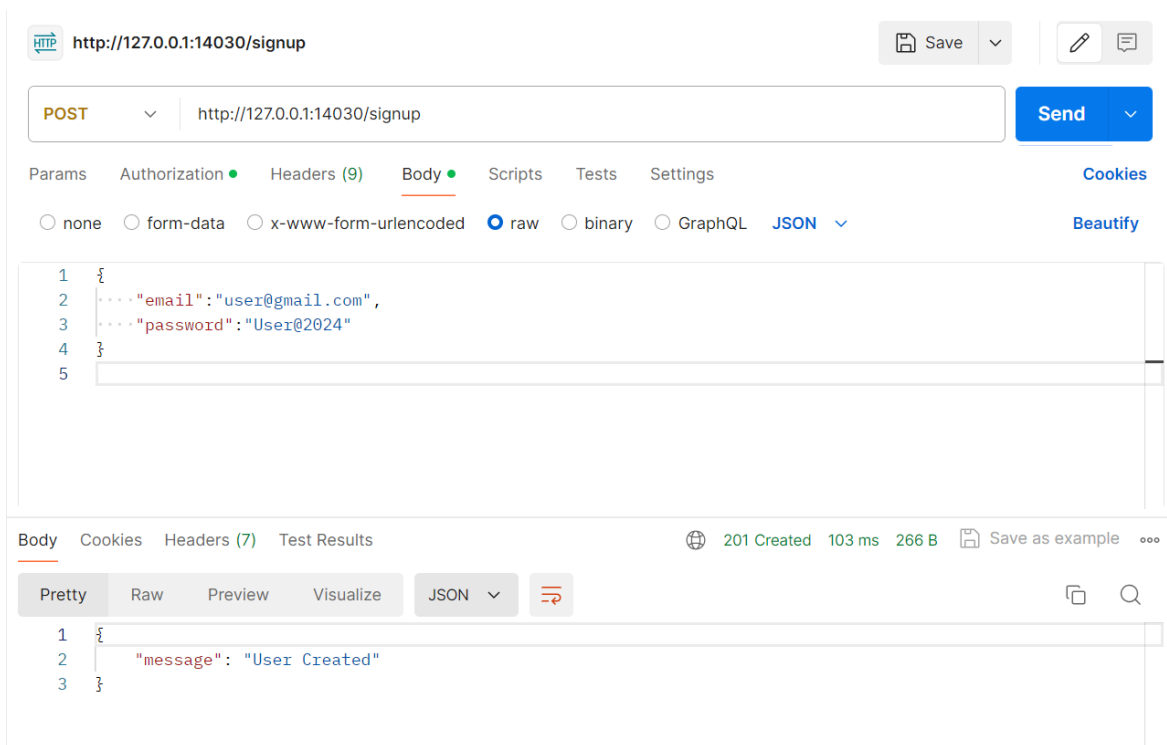
```
spec:
  containers:
    - name: users-api
      image: ndambuki/kubernetes-user-api:latest
```



```
env:
  - name: AUTH_ADDRESS
    value: "auth-service.default"
```

We will just change the value of the AUTH\_ADDRESS.

The API still works



image\_55.png

## Special Environment Variables

There a certain pattern of writing environment variable that is recognized by kubernetes.

```
const response = await
axios.get(`http://${process.env.AUTH_SERVICE_SERVICE_HOST}/hash/${password
}`);
```

This pattern has:

```
<SERVICE_NAME>_SERVICE_HOST
```

Even our User service will have the same.

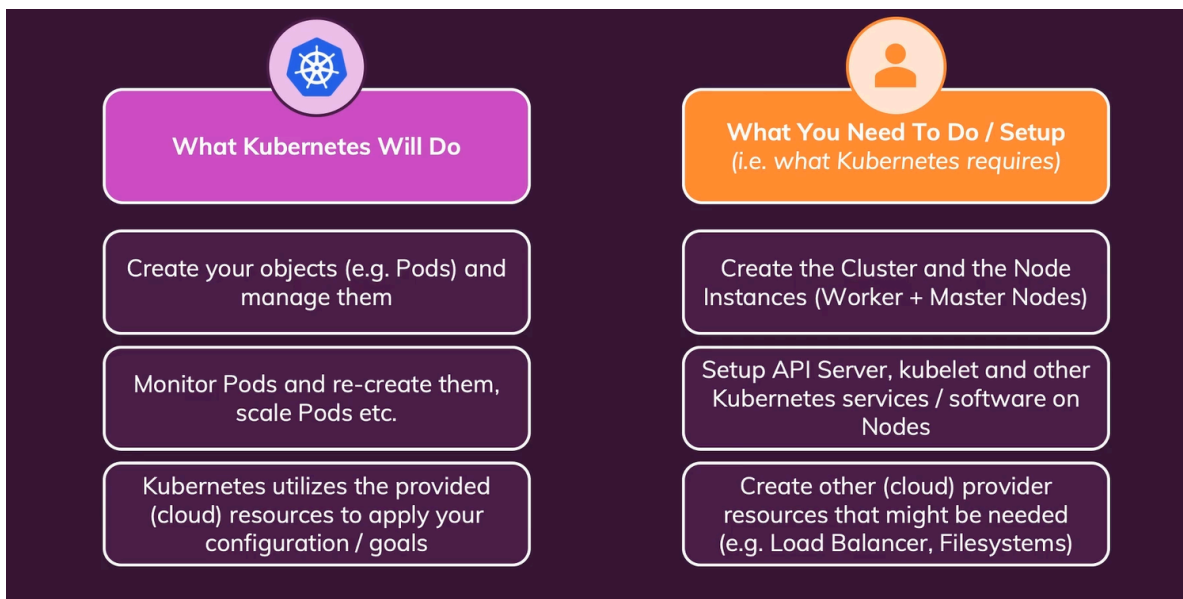
# Deployment

In this module we will be deploying a Full-stack application to an EKS.

## EKS

Amazon Elastic Kubernetes Service (Amazon EKS) is a managed Kubernetes service that makes it easy for you to run Kubernetes on AWS and on-premises.

But before we dive into deployment . Lets look at the role we play and what kubernetes will help with



image\_28.png

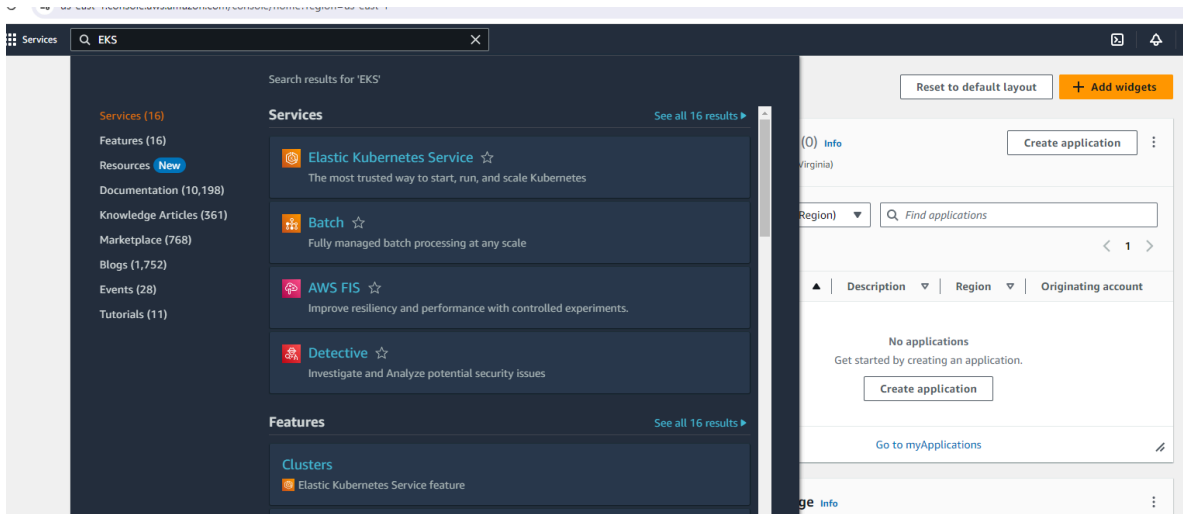
we will start with the database and backend app which can be found at:

[https://github.com/joe6276/Kubernetes-](https://github.com/joe6276/Kubernetes-101/tree/master/Kubernetes%20Projects/4.%20Kubernetes-%20Deployment)

[101/tree/master/Kubernetes%20Projects/4.%20Kubernetes-%20Deployment](https://github.com/joe6276/Kubernetes-101/tree/master/Kubernetes%20Projects/4.%20Kubernetes-%20Deployment)

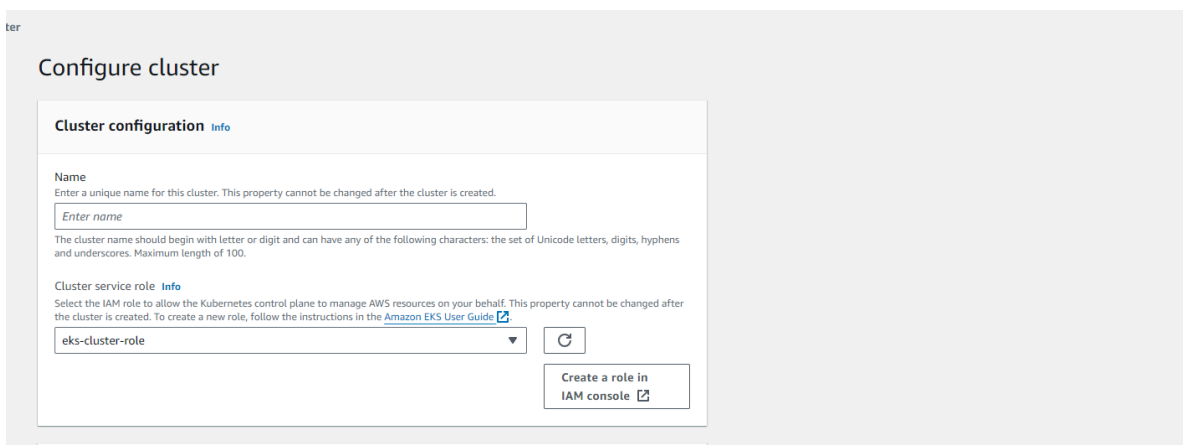
First we will create a Cluster at AWS

## Search and Click on EKS



image\_29.png

Then Click on add Cluster - Enter the Cluster Name



image\_30.png

Then we need to give Kubernetes some permissions. so click on create a role

**On select trusted entity**

Step 1  
**Select trusted entity**

Step 2  
Add permissions

Step 3  
Name, review, and create

## Select trusted entity [info](#)

**Trusted entity type**

- ☒ **AWS service**  
Allow AWS services like EC2, Lambda, or others to perform actions in this account.
- ☐ **AWS account**  
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.
- ☐ **Web identity**  
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.
- ☐ **SAML 2.0 federation**  
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.
- ☐ **Custom trust policy**  
Create a custom trust policy to enable others to perform actions in this account.

**Use case**  
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case  
EKS

Choose a use case for the specified service.  
Use case

- ☐ **EKS**  
Allows EKS to manage clusters on your behalf.
- ☒ **EKS - Cluster**  
Allows access to other AWS service resources that are required to operate clusters managed by EKS.
- ☐ **EKS - Nodegroup**  
Allow EKS to manage nodegroups on your behalf.
- ☐ **EKS - Fargate pod**  
Allows access to other AWS service resources that are required to run Amazon EKS pods on AWS Fargate.
- ☐ **EKS - Fargate profile**  
Allows EKS to run Fargate tasks.


image\_31.png

Leave the defaults and click on next

## Add Permissions

## Add permissions [info](#)

**Permissions policies (1) [info](#)**  
The type of role that you selected requires the following policy.

Policy name <a href="#">🔗</a>	Type
 <a href="#">AmazonEKSClusterPolicy</a>	AWS managed

► Set permissions boundary - *optional*

Cancel Previous **Next**

image\_32.png

Here we need the Eks Cluster policy

## Name Review and Create

Name, review, and create

Role details

**Role name**  
Enter a meaningful name to identify this role.

Maximum 64 characters. Use alphanumeric and '+, -, @, \_' characters.

**Description**  
Add a short explanation for this role.

Maximum 1000 characters. Use letters (A-Z and a-z), numbers (0-9), tabs, new lines, or any of the following characters: \_ + , @ - / [ ] # \$ % ^ \* & ' " ~

**Step 1: Select trusted entities** Edit

Trust policy

```

1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "Allow",
6       "Principal": {
7         "Service": [
8           "eks.amazonaws.com"
9         ]
10      },
11      "Action": "sts:AssumeRole"
12    ]
13  }
14 }
```

image\_33.png

give it a name and review the rest

Then create the role

Now use the role we have created:

Cluster service role [Info](#)

Select the IAM role to allow the Kubernetes control plane to manage AWS resources on your behalf. This property cannot be changed after the cluster is created. To create a new role, follow the instructions in the [Amazon EKS User Guide](#).

▼

↺

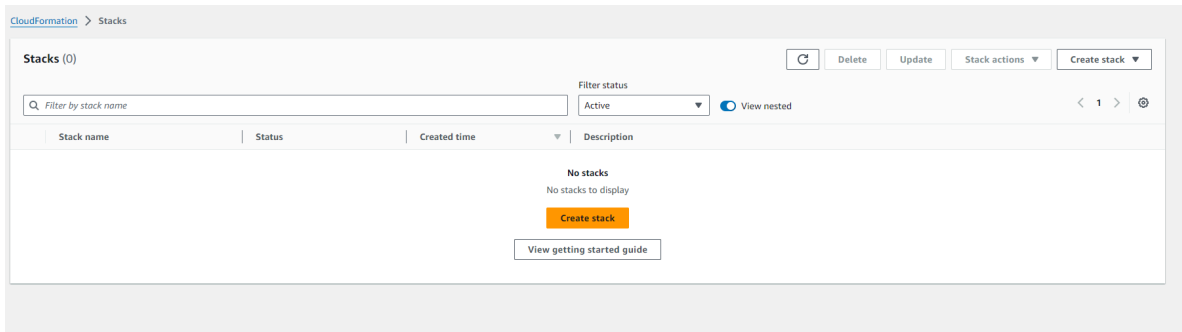
Create a role in IAM console [↗](#)

image\_34.png

Click on next

## Cluster Networking

on the VPC duplicate the Tab and on your AWS Search Cloud Formation



image\_35.png

click on create stack

Paste this link on the Amazon S3 URL : <https://s3.us-west-2.amazonaws.com/amazon-eks/cloudformation/2020-10-29/amazon-eks-vpc-private-subnets.yaml>

image\_36.png

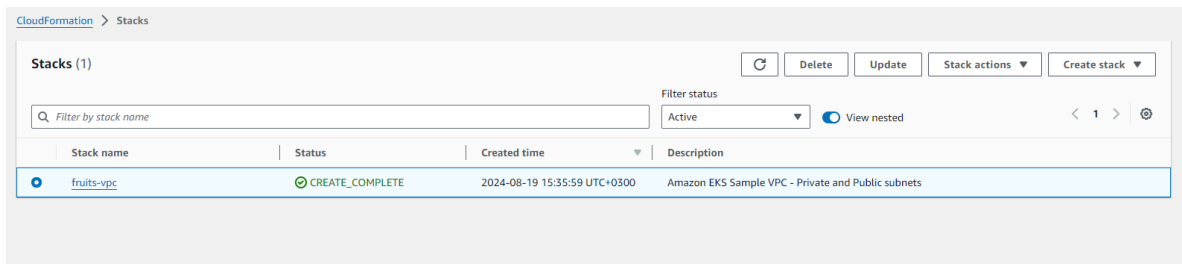
click on Next

image\_37.png

give it a name

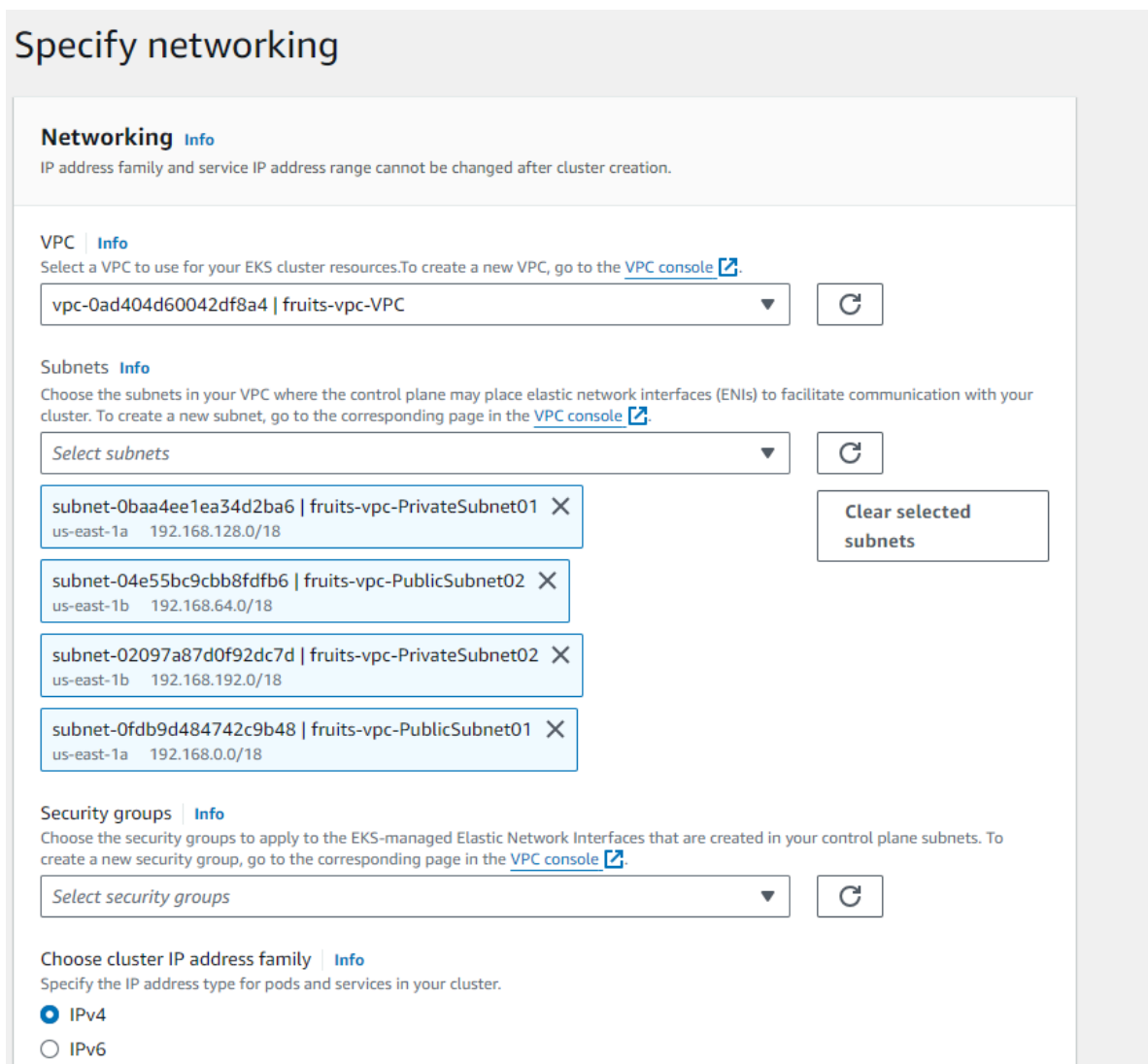
For the next pages leave the defaults and submit to create the VPC.

Once the VPC is created:



image\_38.png

Back to the Cluster page select It:



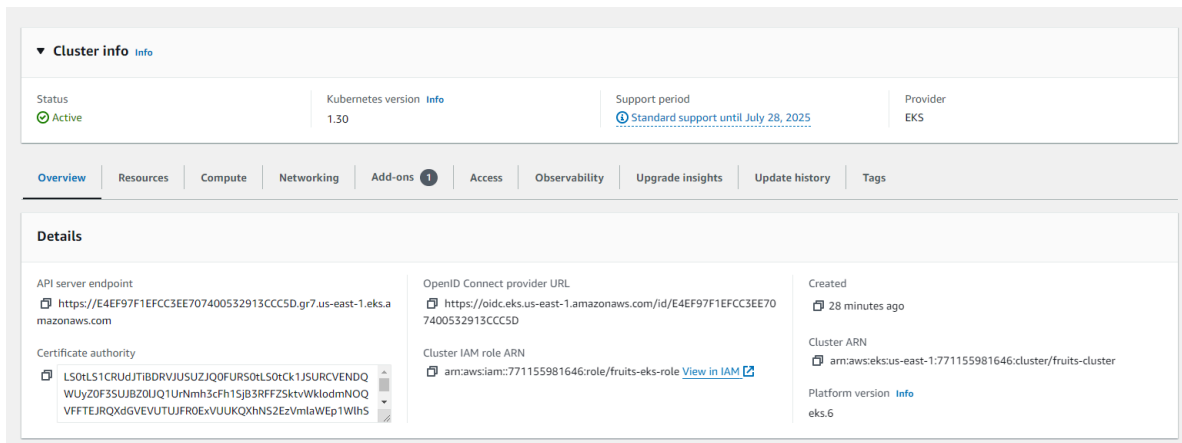
image\_39.png

For the other pages leave the defaults as they are then create the cluster



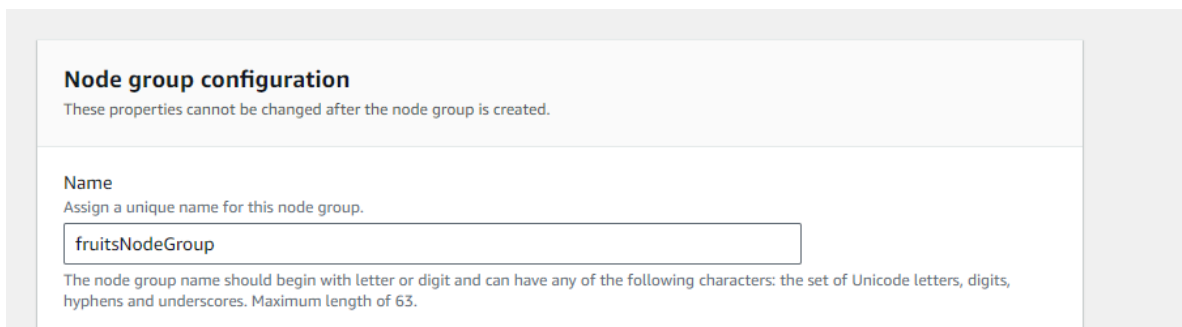
# Creating Worker Nodes

Now let create a worker node:



image\_40.png

click on compute:



image\_41.png

Give it a name

On the Role part click on create role in IAM console:

**select trusted entity**

Select trusted entity [Info](#)

**Trusted entity type**

☒ **AWS service**  
Allow AWS services like EC2, Lambda, or others to perform actions in this account.

☐ **AWS account**  
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.

☐ **Web identity**  
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.

☐ **SAML 2.0 federation**  
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

☐ **Custom trust policy**  
Create a custom trust policy to enable others to perform actions in this account.

**Use case**  
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case  
EC2 ▼

Choose a use case for the specified service.  
Use case

☒ **EC2**  
Allows EC2 instances to call AWS services on your behalf.

☐ **EC2 Role for AWS Systems Manager**  
Allows EC2 instances to call AWS services like CloudWatch and Systems Manager on your behalf.

☐ **EC2 Spot Fleet Role**  
Allows EC2 Spot Fleet to request and terminate Spot instances on your behalf.

☐ **EC2 - Spot Fleet Auto Scaling**  
Allows Auto Scaling to access and update EC2 spot fleets on your behalf.

image\_42.png

## Add permissions

Then select these 3 permissions Add the newly created role:

## Configure node group [Info](#)

A node group is a group of EC2 instances that supply compute capacity to your Amazon EKS cluster. You can add multiple node groups to your cluster.

### Node group configuration

These properties cannot be changed after the node group is created.

#### Name

Assign a unique name for this node group.

fruitsNodeGroup

The node group name should begin with letter or digit and can have any of the following characters: the set of Unicode letters, digits, hyphens and underscores. Maximum length of 63.

#### Node IAM role [Info](#)

Select the IAM role that will be used by the nodes. To create a new role, go to the [IAM console](#).

fruits-roles



The selected role must not be used by a self-managed node group as this could lead to a service interruption upon managed node group deletion.

[Learn more](#)

Create a role in  
IAM console [↗](#)

### Launch template [Info](#)

These properties cannot be changed after the node group is created.



Use launch template

Configure this node group using an EC2 launch template.

image\_43.png

## Compute and Scaling Configuration

## Set compute and scaling configuration

### Node group compute configuration

These properties cannot be changed after the node group is created.

#### AMI type [Info](#)

Select the EKS-optimized Amazon Machine Image for nodes.

Amazon Linux 2 (AL2\_x86\_64) ▼

#### Capacity type

Select the capacity purchase option for this node group.

On-Demand ▼

#### Instance types [Info](#)

Select instance types you prefer for this node group.

Q Enter an instance type

t3.medium

vCPU: 2 vCPUs Memory: 4 GiB Network: Up to 5 Gigabit Max ENI: 3 Max IPs: 18

#### Disk size

Select the size of the attached EBS volume for each node.

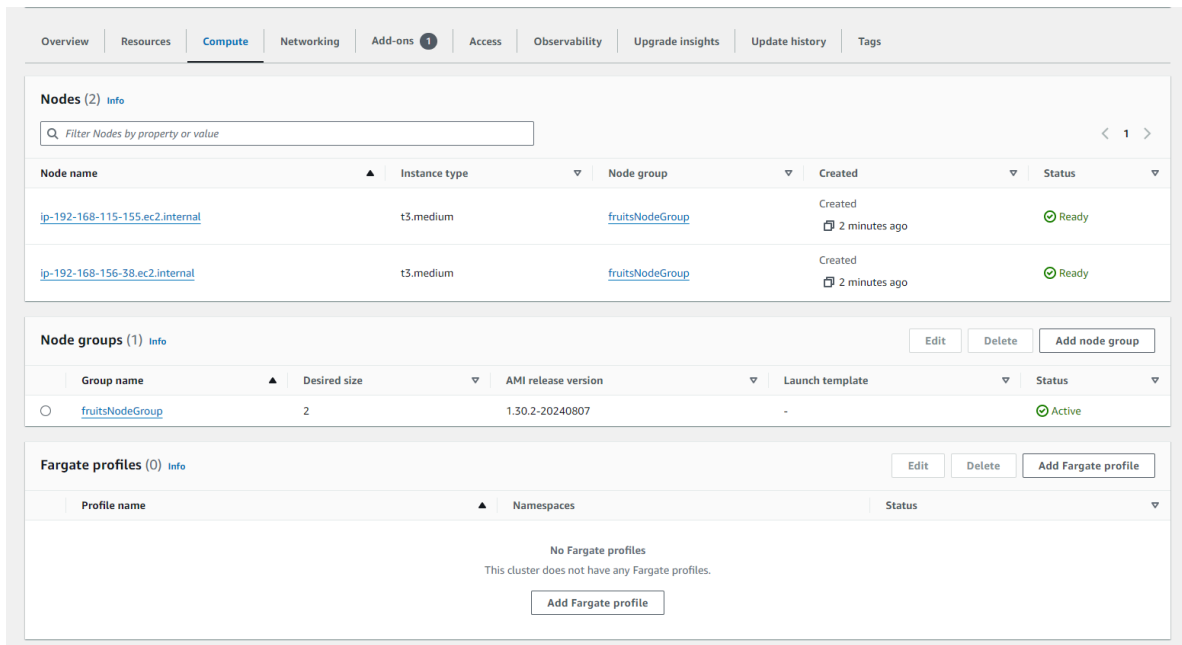
20

GiB

image\_44.png

Here select the specification of the EC2 instances you want to use. Also select on the Node group scaling configuration (or go with the default):





image\_46.png

Now let's create the configuration .

## Database Service

```
apiVersion: v1
kind: Service
metadata:
  name: database-service
spec:
  selector:
    app: database
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 1433
      targetPort: 1433
```

## Database Deployment

```
apiVersion: apps/v1
kind: Deployment
```

```

metadata:
  name: database-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: database
  template:
    metadata:
      labels:
        app: database
    spec:
      containers:
        - name: database
          image: mcr.microsoft.com/mssql/server:2019-latest
          env:
            - name: MSSQL_SA_PASSWORD
              value: Root@2024
            - name: ACCEPT_EULA
              value: 'Y'

```

## Fruit Backend Service

```

apiVersion: v1
kind: Service
metadata:
  name: fruit-service
spec:
  selector:
    app: fruit
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80

```

## Fruit Backend Deployment

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: fruit-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fruit
  template:
    metadata:
      labels:
        app: fruit
    spec:
      containers:
        - name: fruit
          image: ndambuki/kub-fruit:latest
          env:
            - name: DB_USER
              value: 'sa'

            - name: DB_PWD
              value: 'Root@2024'

            - name: DB_NAME
              value: 'Fruits'

```

Before we can apply the changes:

let's configure our .kube config.file (Copy the backup somewhere )

First configure your AWS: here (<https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html>)

Then Run this command:

```

aws eks --region <your region> update-kubeconfig --name <your cluster
name>

```



E.g. My configuration

```
aws eks --region us-east-1 update-kubeconfig --name fruits-cluster
```

Now run the configuration:

```
kubectl apply -f database-service.yaml -f database.yaml -f fruit-  
service.yaml -f fruit.yaml
```

Now everything should be working well:

```
netes> kubectl get pods  
NAME                                READY   STATUS    RESTARTS   AGE  
database-deployment-5d6b44f5d5-g89nv 1/1     Running   0           98s  
fruit-deployment-6b7987cd5f-55nb4    1/1     Running   0           97s
```

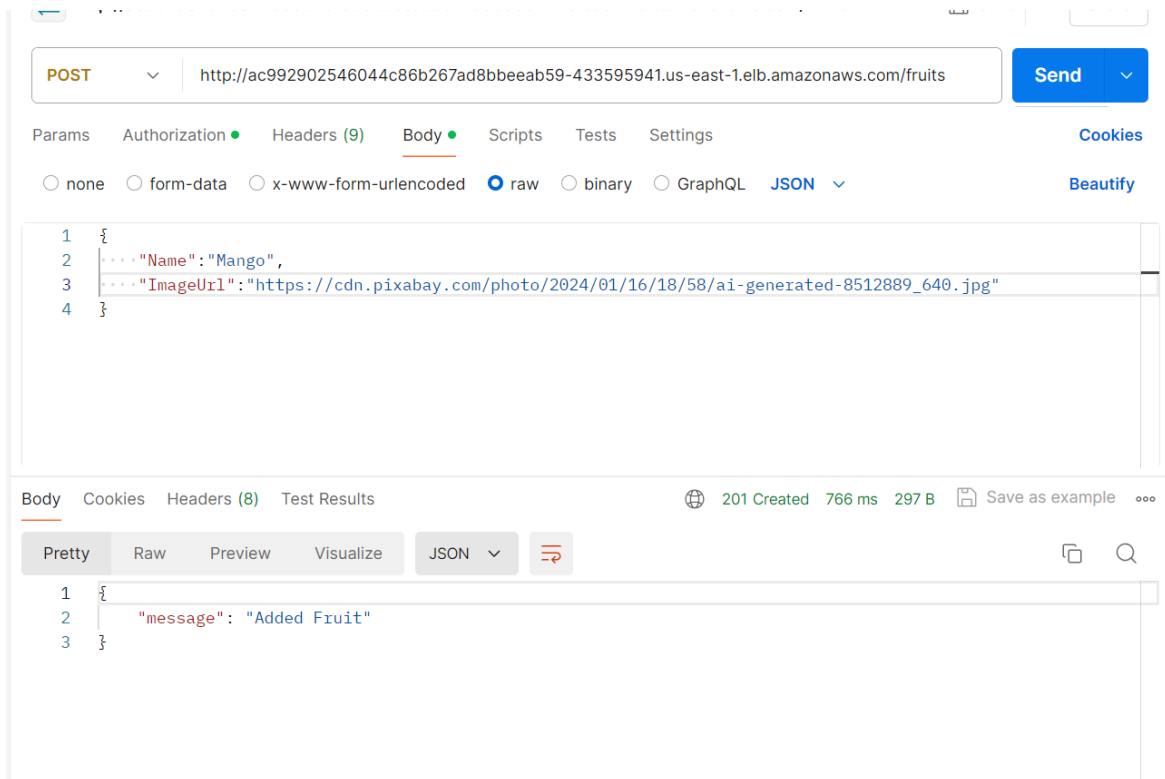
image\_47.png

Then the services

```
netes> kubectl get services  
NAME          TYPE          CLUSTER-IP      EXTERNAL-IP  
database-service ClusterIP      10.100.34.43    <none>  
1433/TCP      2m44s  
fruit-service LoadBalancer   10.100.171.212  ac992902546044c86b267ad8bbeab59-433595941.us-east-1.elb.amazonaws.com  
80:30825/TCP  2m42s  
kubernetes    ClusterIP      10.100.0.1      <none>  
443/TCP       104m
```

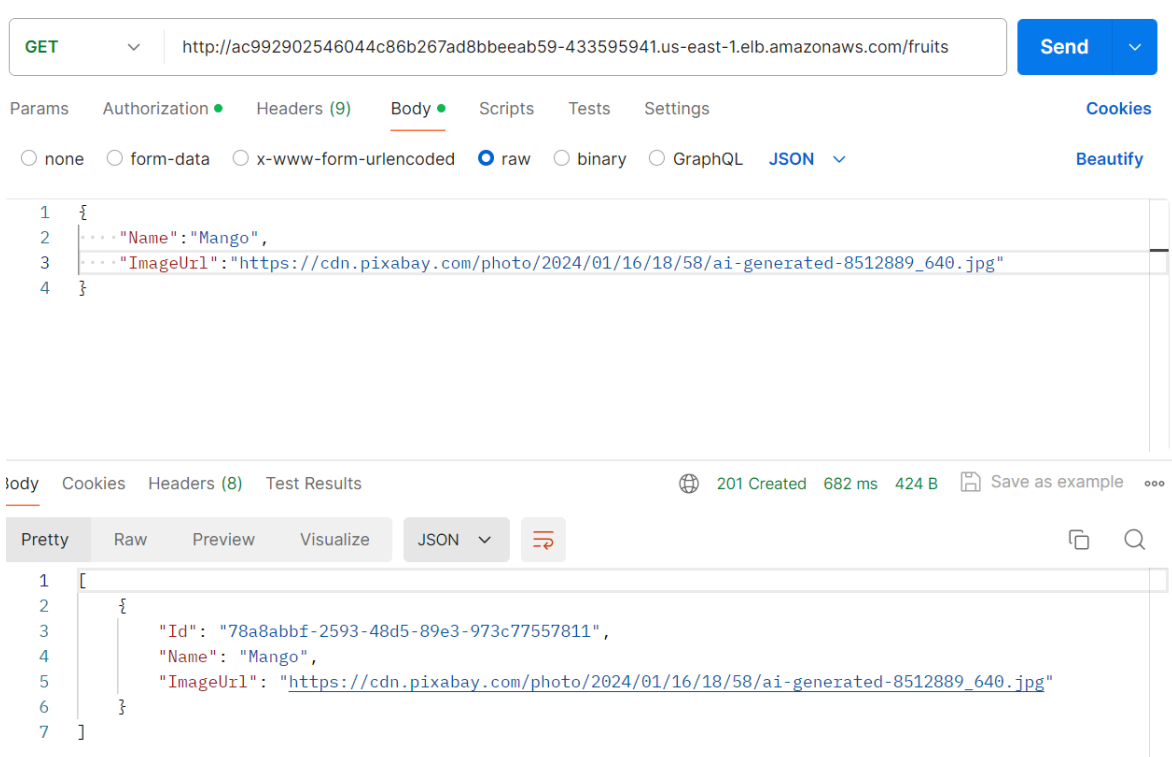
image\_48.png

The loadBalancer can now run in postman or the browser e.g.:




image\_49.png

You can also get the added fruits:



image\_50.png

 On your frontend connect to the backend LoadBalancer then push the image

## Frontend Service

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

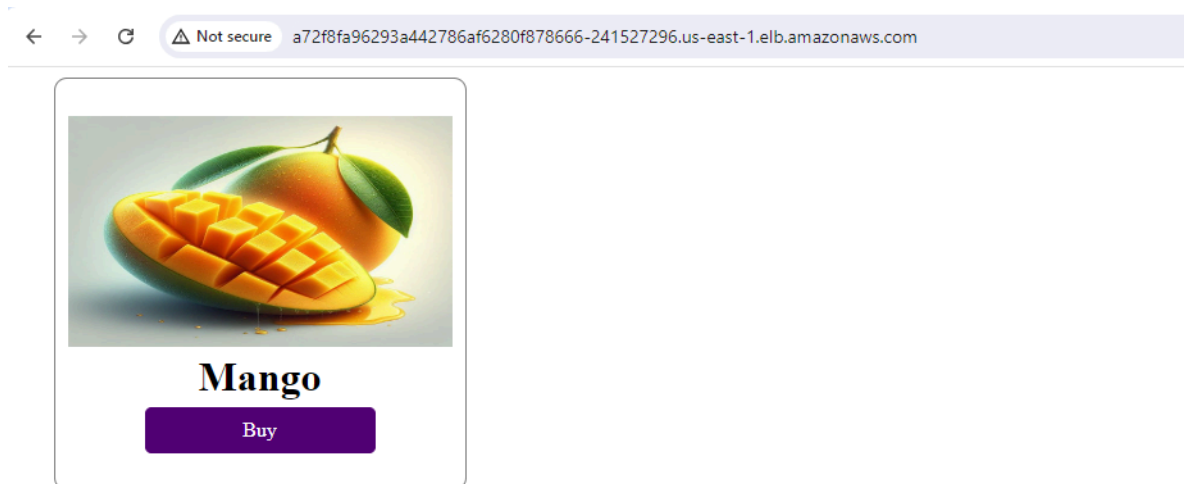
## Frontend Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend
          image: ndambuki/kub-fruit-frontend:latest
```

apply these Changes

```
kubectl apply -f frontend.yaml -f frontend-service.yaml
```

And the Application is Live



image\_51.png