Lab 5: Kubernetes

Scenario based – Robots City

Docker addresses the problem of "works on my machine but not on yours" by packaging applications together with their dependencies into a single, portable image. This allows developers to deliver a Docker image to DevOps engineers with confidence that the application will behave consistently across different environments.

Now consider the perspective of a DevOps engineer. A developer has provided a Docker image for the company's website, which must be deployed to serve millions of users. Due to the high volume of traffic, the application cannot be hosted on a single machine and instead must be deployed across multiple machines to distribute the load effectively.

At this point, the challenge is no longer about building the container, but about managing and running containers across all these machines. One possible approach is to manually connect to each virtual machine and execute Docker commands individually, or to automate this process using scripts. While feasible, this approach quickly becomes inefficient and difficult to manage at scale.

Updates to the docker image further increases the complexity. Each new version of the Docker image would require redeployment across all machines. Performing such updates while ensuring that all servers remain available to handle customer traffic especially under high load adds significant operational risk and complexity.

These challenges highlight the limitations of managing containers manually in large-scale environments. A more robust and systematic solution is required to coordinate container deployment, scaling, updates, and availability across many machines. Kubernetes, often abbreviated as k8s, provides this solution. Kubernetes is a **container orchestration platform** designed to automate the deployment, management, scaling, and operation of containerized applications across distributed systems.

Prerequisites:

1. Make sure docker is running
2. Run **minikube start** to start using minikube

NOTE: Make sure to use SRN wherever applicable properly, do not use as pes1ug23csxxx.

Introduction:

Kubernetes cluster – entire city

Kubernetes works as a **cluster**, which consists of two or more machines working together to perform a shared task. In a standard Kubernetes setup, the cluster is composed of multiple **nodes**, where one node acts as the **control plane** and the remaining nodes act as **worker nodes**. The control plane runs components responsible for managing and coordinating the behaviour of all nodes within the cluster.

Command to see how many nodes we have in our minikube:

kubectl get nodes

Task 1: Creating Namespace

1. Namespace – districts of the city

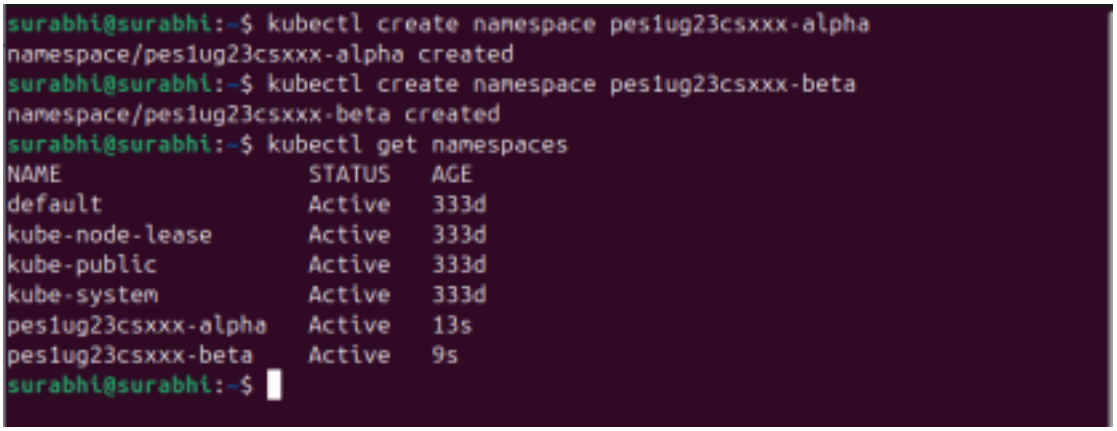
Just as how districts partition a city into sub-cities, namespaces partition clusters into virtual sub-clusters. The names of the resources/objects created inside a Namespace are unique, but not across Namespaces in the cluster.

• Creating districts/namespaces:

kubectl create namespace pes1ug23csxxx-alpha
kubectl create namespace pes1ug23csxxx-beta

• To list all namespaces:

kubectl get namespaces



Generally, Kubernetes creates four Namespaces by default: **kube-system, kube public, kube-node-lease,** and **default.**

• The **kube-system** Namespace contains the objects created by the Kubernetes system, mostly the control plane agents.

•The **default** Namespace contains the objects and resources created by administrators and developers, and objects are assigned to it by default unless another Namespace name is provided by the user.

• **kube-public** is a special Namespace, which is unsecured and readable by anyone, used for special purposes such as exposing public (non-sensitive) information about the cluster.

• The newest Namespace is **kube-node-lease** which holds node lease objects used for node heartbeat data.

kubectl creates all objects in the default namespace unless mention explicitly using the below command:

kubectl config set-context --current --namespace=pes1ug23csxxx-alpha

2. Pods – Citizens of the city

A Pod is the smallest Kubernetes workload object. It is the unit of deployment in Kubernetes, which represents a single instance of the application. A Pod is a logical collection of one or more containers, enclosing and isolating them to ensure that they are scheduled together on the same host with the Pod and have access to mount the same

external storage (volumes) and other common dependencies.

A pod is like a citizen. Citizens can get sick, disappear, or multiply. • Creating pod in namespace pes1ug23csxxx-alpha:

**kubectl run citizen **
**--image=nginx **
--restart=Never

--restart=Never tells kubectl to create a pod (kubectl run by default creates a deployment)

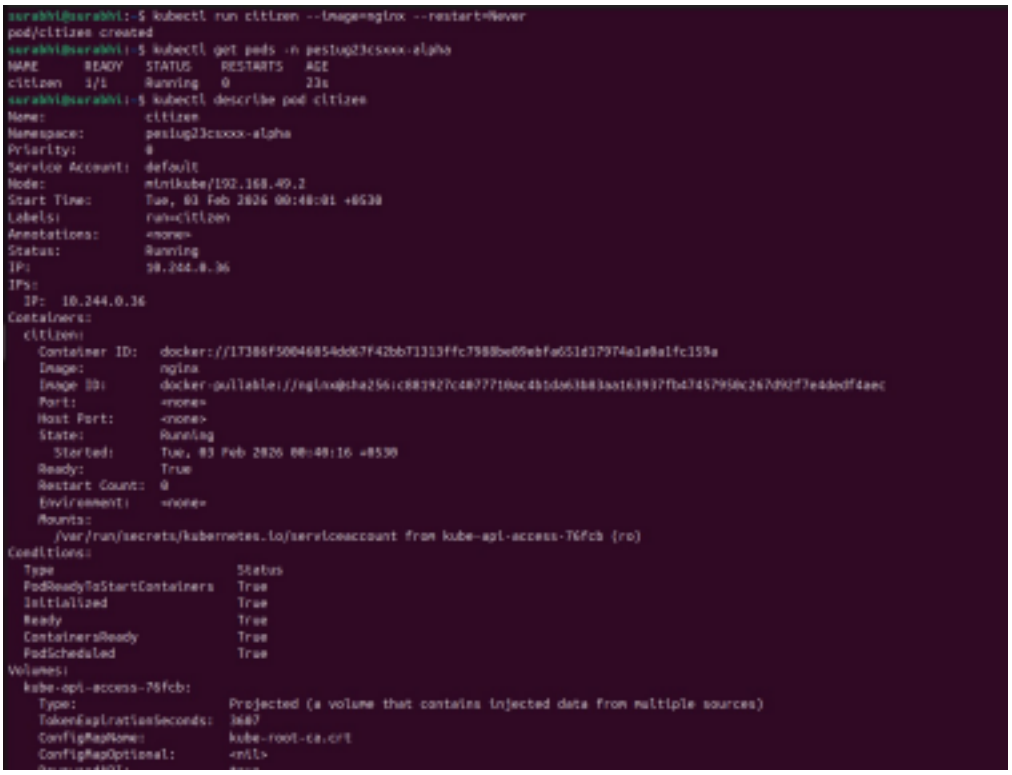
Verification:

To view all pods in a namespace

kubectl get pods

To view more details about a specific pod in a namespace:

kubectl describe pod citizen



Deleting a pod:

kubectl delete pod citizen

3. Watch mode - City surveillance cameras (-w)

Live watching in Kubernetes allows you to observe changes to cluster resources in real time. By using the --watch (or -w) option with kubectl commands, Kubernetes continuously streams updates whenever the state of a resource changes, such as when pods are created, deleted, or restarted. This is especially useful during deployments, scaling operations, and troubleshooting, as it provides immediate feedback on how the system responds to changes.

In terminal 1:

kubectl get pods -w

In terminal 2:

kubectl run temp --image=nginx

kubectl delete pod temp

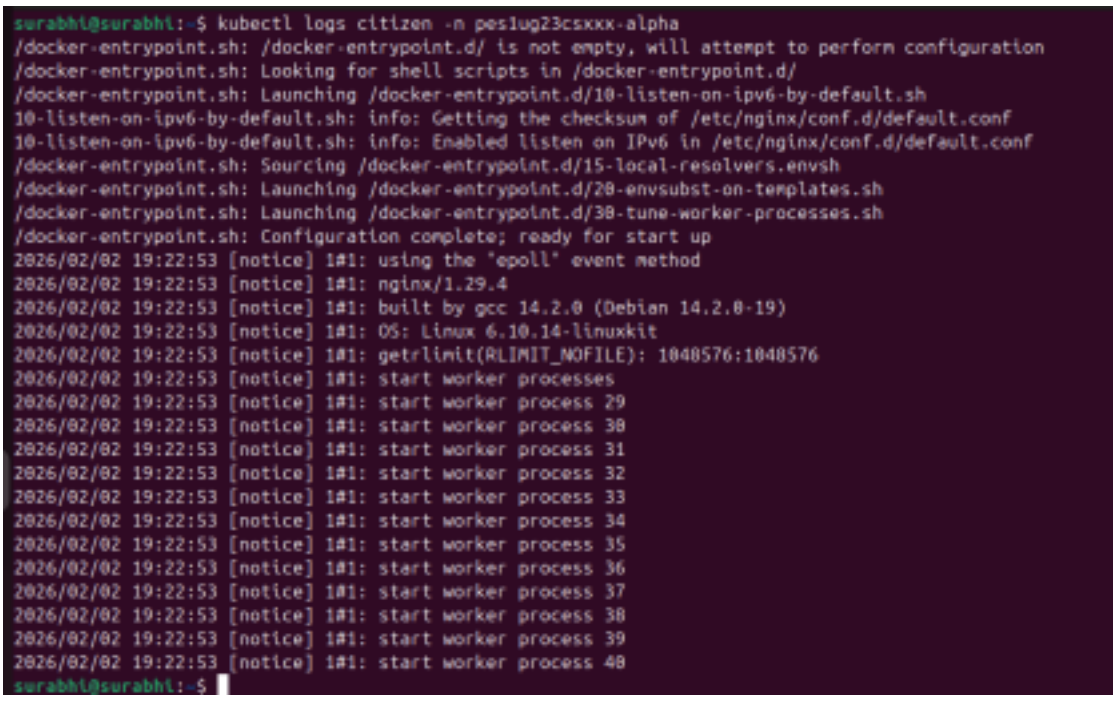
screenshot of terminal 1:

4. Logs – Diaries of citizens

Create pod again using the command given before.

To view the logs of a pod:

kubectl logs citizen



crashing a pod and then checking its logs:

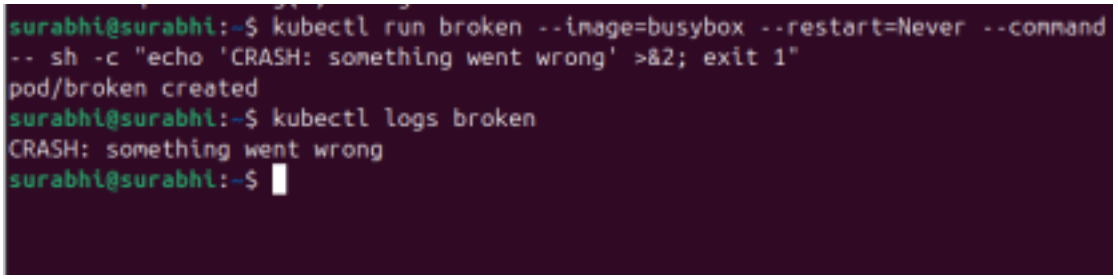
**kubectl run broken **

**--image=busybox **

**--restart=Never **

--command -- sh -c "echo 'CRASH: something went wrong' >&2; exit 1"

kubectl logs broken



This command shows writing custom data into logs. '>&2' means writing into stderr. --command flag tells the pod to run the command mentioned after the flag.

5. Deployments – Factories that produce the citizen robots

Running a single application instance (pod) introduces the risk of service disruption due to application or node failures. To improve reliability and availability, multiple instances of an application can be run in parallel. In Kubernetes, this is achieved using a **ReplicaSet**, which ensures that a specified number of pod replicas are running at all times and supports both manual and automatic scaling.

In practice, ReplicaSets are managed indirectly through **Deployments**. A Deployment is a higher-level abstraction that creates and controls a ReplicaSet, which in turn manages the underlying pods. Deployments provide all the functionality of ReplicaSets along with additional features such as rolling updates and version management.

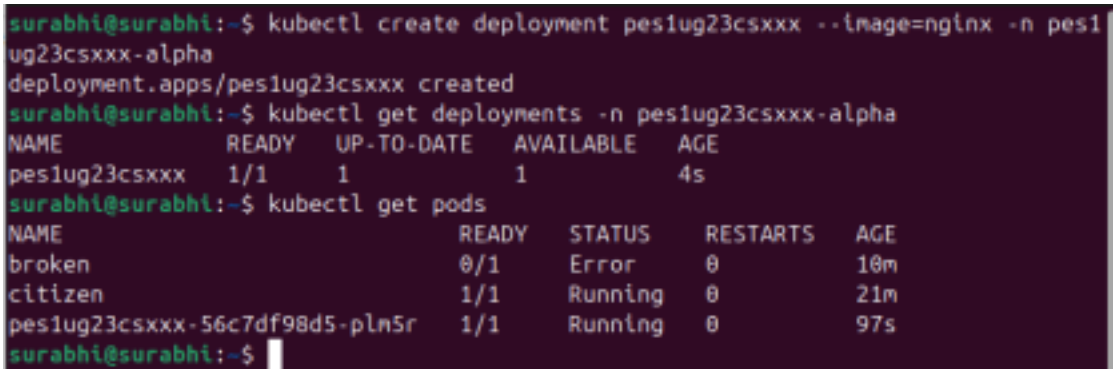
Deployments produce pods reliably

Creating deployments:

**kubectl create deployment pes1ug23csxxx **
--image=nginx

Verification:

kubectl get deployments -n pes1ug23csxxx-alpha
kubectl get pods -n pes1ug23csxxx-alpha



notice the new pod produced by the deployment.

6. Scaling – to grow robot civilisation

Scaling a Deployment in Kubernetes adjusts the number of pod replicas running an application to handle changes in load or availability requirements.

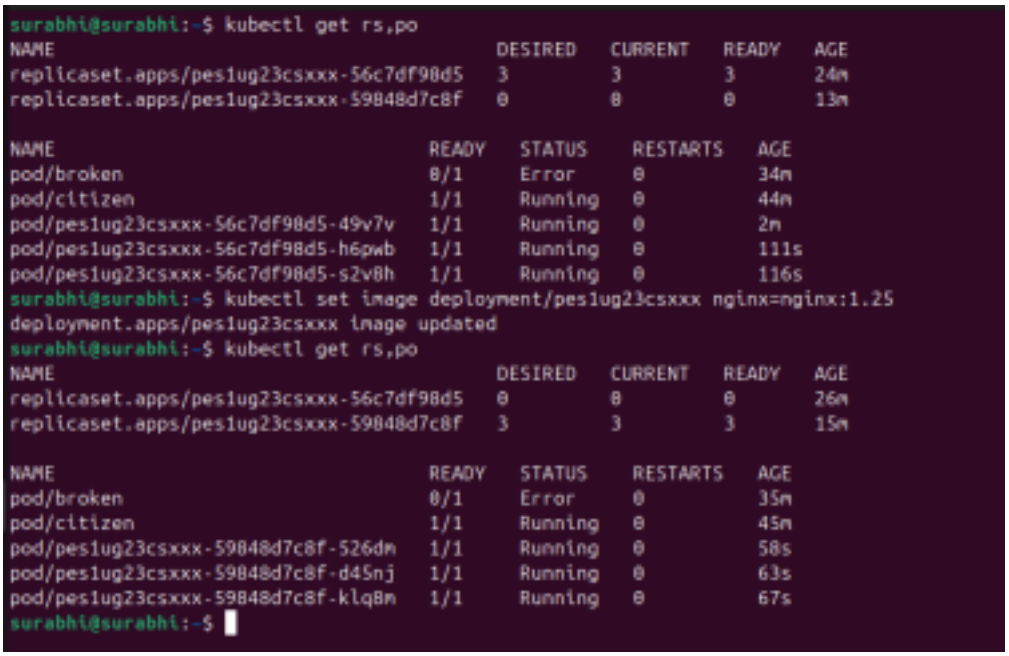
Scaling deployment:
kubectl scale deployment pes1ug23csxxx --replicas=5
Verification:
kubectl get pods

Scaling down:
kubectl scale deployment pes1ug23csxxx --replicas=3
kubectl get pods

```
surabhi@surabhi:~$ kubectl scale deployment pes1ug23csxxx --replicas=5
deployment.apps/pes1ug23csxxx scaled
surabhi@surabhi:~$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
broken                            0/1     Error     0          25m
citizen                           1/1     Running   0          36m
pes1ug23csxxx-56c7df98d5-gbms2    1/1     Running   0          20s
pes1ug23csxxx-56c7df98d5-j8tbw    1/1     Running   0          20s
pes1ug23csxxx-56c7df98d5-kw6zk    1/1     Running   0          4m8s
pes1ug23csxxx-56c7df98d5-rjlj4    1/1     Running   0          20s
pes1ug23csxxx-56c7df98d5-z8w94    1/1     Running   0          20s
surabhi@surabhi:~$ kubectl scale deployment pes1ug23csxxx --replicas=3
deployment.apps/pes1ug23csxxx scaled
surabhi@surabhi:~$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
broken                            0/1     Error     0          25m
citizen                           1/1     Running   0          36m
pes1ug23csxxx-56c7df98d5-j8tbw    1/1     Running   0          30s
pes1ug23csxxx-56c7df98d5-kw6zk    1/1     Running   0          4m18s
pes1ug23csxxx-56c7df98d5-z8w94    1/1     Running   0          30s
surabhi@surabhi:~$
```

7. Rolling updates – renovating city
Renovating city without kicking citizens out.

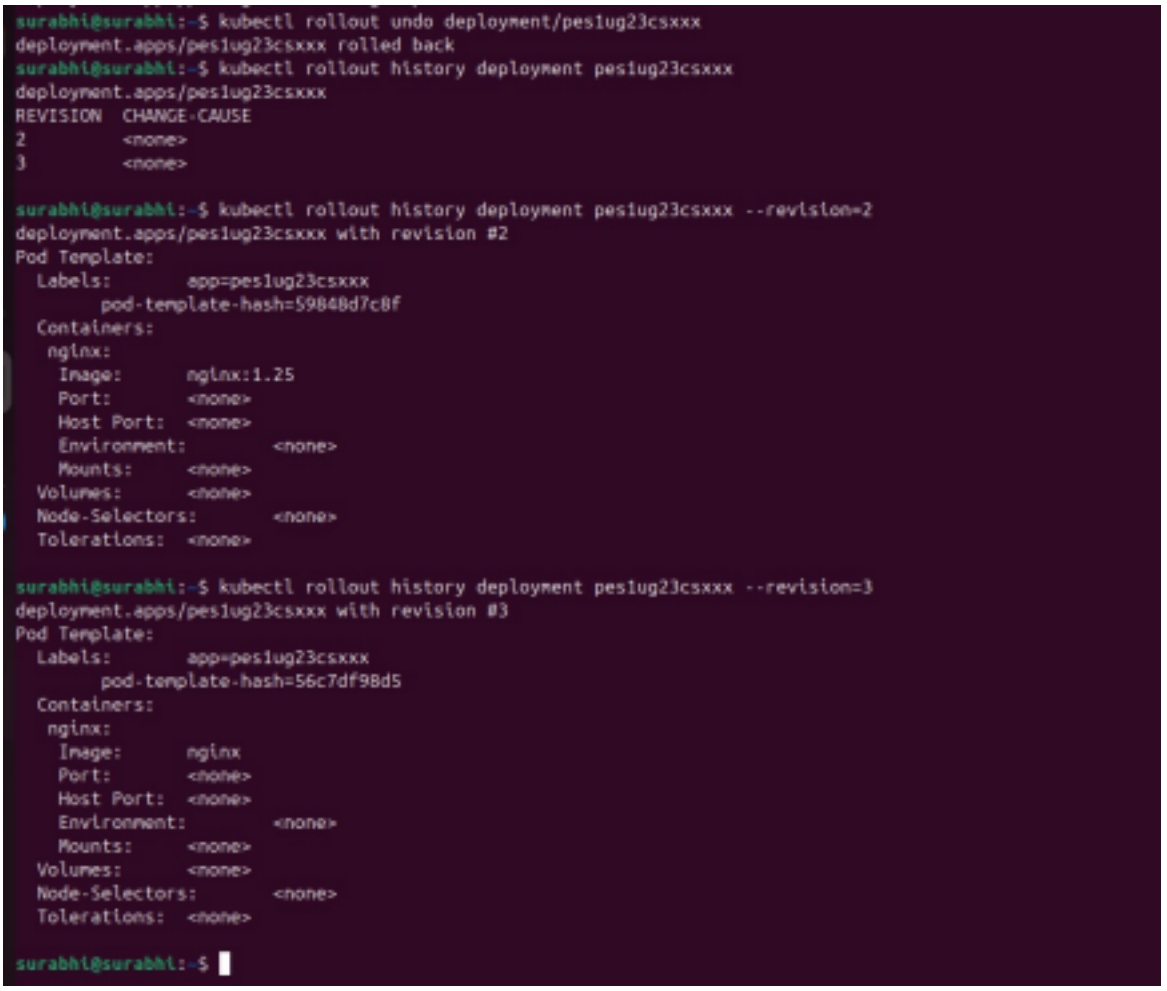
Rolling updates in Kubernetes allow an application to be updated (eg. Updating its image) without downtime by gradually replacing old pods with new ones. Instead of stopping all instances at once, Kubernetes updates a few pods at a time. This ensures that the application remains available to users during the update process. If an issue occurs, the update can be paused or rolled back. Rolling updates help deploy new versions safely and reliably.
Run :
kubectl get rs,po (meaning get replica sets and pods)
Update image used by deployement:
**kubectl set image deployment/pes1ug23csxxx **
nginx=nginx:1.25
kubectl get rs,po
(wait for sometime for the new pods to get created)



Notice how new pods get created.
The way the image update happens is the Deployment controller creates a new ReplicaSet for nginx:1.25 and the old ReplicaSet is scaled down to 0 pods when the new ReplicaSet's pods are running.

Suppose we want the old version of the image back, then run:
kubectl rollout undo deployment/pes1ug23csxxx

To view the revisions made:
kubectl rollout history deployment pes1ug23csxxx (to get revision numbers)
kubectl rollout history deployment pes1ug23csxxx –revision=<revision number>



8. Applying config files + deploying custom docker image on cluster– creating blueprint of city

   Since deployments can have various configurations or flags that need to be set, it is difficult to type them all in a terminal interface. Thus, kubernetes uses configuration files. These are .yaml files which can be used to create pods, deployments and services. Each configuration file has 3 components:

   a. Metadata
   b. Specification (under “spec:”)
   c. Status – Automatically generated and edited by Kubernetes. This status compares the desired and actual states of the components and fixes it in case of a difference between the two. This is part of the self-healing feature of Kubernetes.

   So in case of any updates to a cluster, instead of running a huge set of commands again we can just run the updates configuration file once.

   i. Create project folder:

   mkdir PES1UG23CSXXX

   cd PES1UG23CSXXX

   ii. Create simple html page:

   mkdir html

   nano html/index.html

index.html:

<!DOCTYPE html>

<html>

<head>

<title>My Robot City</title>

</head>

<body>

<h1>Hello from district PES1UG23CSXXX-beta running in Kubernetes</h1>

</body>

</html>

ctrl+o, enter, ctrl+x to save changes to file

iii. Create dockerfile

nano Dockerfile

FROM nginx:alpine

COPY html /usr/share/nginx/html

iv. Run:

eval $(minikube docker-env)

This command points the host terminal to minikube's docker daemon so that any docker images get built inside minikube and not on host docker. This way kubectl gets access to custom docker image.

v. Build docker image:

docker build -t pes1ug23csxxx-nginx:1.0 .

vi. Create .yaml file

nano nginx-pes1ug23csxxx.yaml

nginx-pes1ug23csxxx.yaml file:

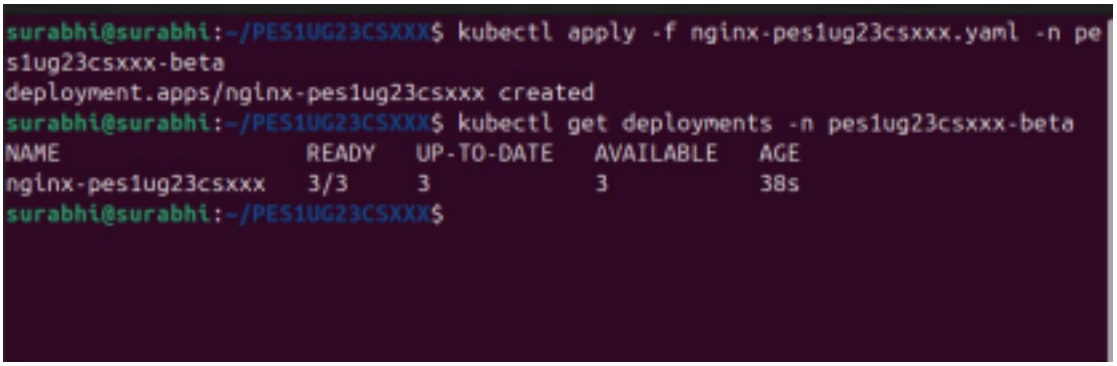
```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-pes1ug23csxxx
spec:
 replicas: 3
 selector:
 matchLabels:
 app: nginx
```

template:
metadata:
labels:
app: nginx
spec:
containers:
- name: nginx
image: pes1ug23csxxx-nginx:1.0
imagePullPolicy: Never
ports:
- containerPort: 80

This configuration file defines a **Deployment** named **nginx-pes1ug23csxxx** that manages an application running the nginx container image pulled from minikube daemon. It specifies that three replicas of the application should always be running to ensure availability. The selector and labels ensure that the Deployment correctly identifies and manages its pods. The template section describes how each pod should be created, including the container name and image to run.
ImagePullPolicy basically directs the deployment to use the local image and not try pulling the image from docker hub.

vii. Application of the yaml file:
kubectl apply -f nginx-pes1ug23csxxx.yaml -n pes1ug23csxxx-beta (notice that we are using a different namespace now)
kubectl config set-context --current –namespace=pes1ug23csxxx-beta (setting new namespace as default namespace)

kubectl get deployments (verification)



9. Services – roads in the city
Provides access to pods

To access the application, a user or another application needs to connect to the Pods. As Pods are ephemeral in nature, resources like IP addresses allocated to them cannot be static. Pods could be terminated abruptly or be rescheduled based on existing requirements.
When a pod goes down and a new pod replaces it, the user might lose access to the old pod as the new pod gets a new IP address.

To overcome this situation, Kubernetes provides a higher-level abstraction called **Service**, which logically groups Pods and defines a policy to access them. This grouping is achieved via **Labels** and **Selectors.** The metadata portion of the configuration file consists of labels and the specification part consists of selectors.

Creating service through pes1ug23csxx-service.yaml file:

apiVersion: v1
kind: Service
metadata:
name: pes1ug23csxxx-service
namespace: pes1ug23csxxx-beta
spec:
type: ClusterIP
selector:
app: nginx
ports:
- port: 80 # Service port
targetPort: 80 # Container port

Here, app is the label, web is the value of the label and app=web (or app:web) is the selector.

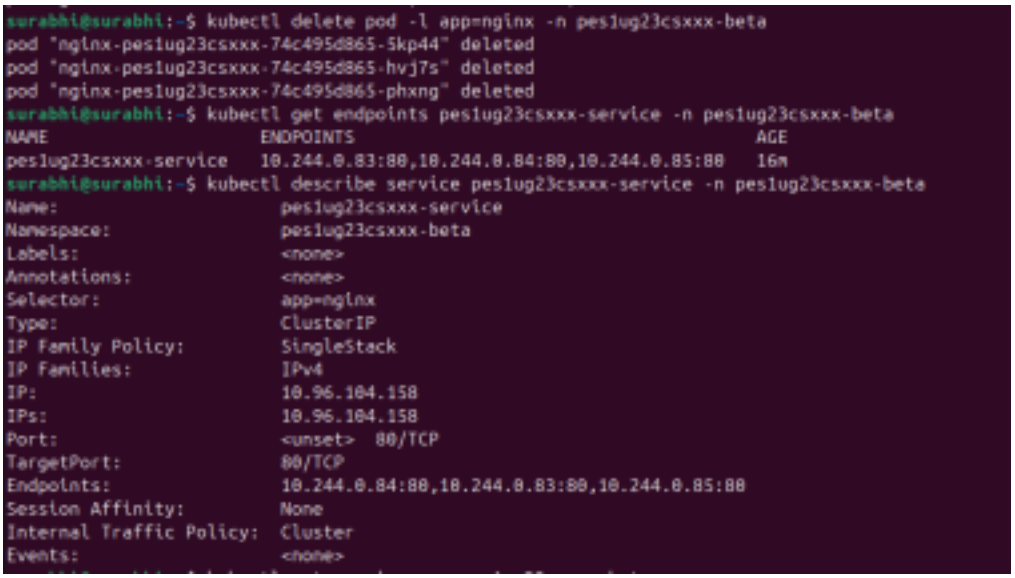
Applying the yaml file:
kubectl apply -f pes1ug23csxxx-service.yaml
verification:
kubectl get services
kubectl describe service pes1ug23csxxx-service
kubectl get endpoints pes1ug23csxxx-service (checking endpoints of pods)

```
surabhi@surabhi:~$ kubectl apply -f pes1ug23csxxx-service.yaml
service/pes1ug23csxxx-service configured
surabhi@surabhi:~$ kubectl get services -n pes1ug23csxxx-beta
NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
pes1ug23csxxx-service   ClusterIP   10.96.104.158   <none>        80/TCP    14m
surabhi@surabhi:~$ kubectl describe service pes1ug23csxxx-service -n pes1ug23csxxx-beta
Name:                     pes1ug23csxxx-service
Namespace:                pes1ug23csxxx-beta
Labels:                   <none>
Annotations:              <none>
Selector:                 app=nginx
Type:                     ClusterIP
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.96.104.158
IPs:                      10.96.104.158
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
Endpoints:                10.244.0.82:80,10.244.0.81:80,10.244.0.80:80
Session Affinity:         None
Internal Traffic Policy:  Cluster
Events:                   <none>
surabhi@surabhi:~$ kubectl get endpoints pes1ug23csxxx-service -n pes1ug23csxxx-beta
NAME                   ENDPOINTS                                            AGE
pes1ug23csxxx-service   10.244.0.80:80,10.244.0.81:80,10.244.0.82:80   14m
surabhi@surabhi:~$
```

Proving traffic routing changes:

Kill pod:
kubectl delete pod -l app=nginx
check endpoints again:
kubectl get endpoints pes1ug23csxxx-service
kubectl describe service pes1ug23csxxx-service
Pods' IPs change but service IP remains same

10. Port Forwarding – tunnels in the city



Port forwarding in Kubernetes allows you to temporarily access an application running inside a cluster by forwarding a local machine port to a port on a pod or service. It is commonly used for testing and debugging, as it provides direct access without exposing the application publicly. The connection exists only while the command is running and is not intended for production use.

kubectl port-forward deployment/nginx-pes1ug23csxxx 8080:80

here we are accessing the nginx app we deployed earlier using our own docker image, so you should see the contents of index.html on localhost:8080

verification: open in browser
http://localhost:8080

11. Cleanup – Evacuate the city

Deleting namespaces clears the entire city:

kubectl delete namespace pes1ug23csxxx-alpha

kubectl delete namespace pes1ug23csxxx-beta

checking (getting all namespaces):

kubectl get ns

```
surabhi@surabhi:~$ kubectl delete namespace pes1ug23csxxx-alpha
namespace "pes1ug23csxxx-alpha" deleted
surabhi@surabhi:~$ kubectl delete namespace pes1ug23csxxx-beta
namespace "pes1ug23csxxx-beta" deleted
surabhi@surabhi:~$ kubectl get ns
NAME              STATUS   AGE
default           Active   333d
kube-node-lease   Active   333d
kube-public       Active   333d
kube-system       Active   333d
```