😊 Pods

In Kubernetes, a pod is the smallest and simplest unit of deployment. It represents a single instance of a running application in the cluster. Pods are the basic building blocks of Kubernetes applications and encapsulate one or more containers, storage resources, and networking configurations.

Here are some key aspects of pods in Kubernetes:

1. **Container(s):** A pod can contain one or more containers, which share the same network namespace, IP address, and storage volumes. Containers within the same pod can communicate with each other over the localhost interface.
2. **Atomic Unit:** Pods are considered as atomic units in Kubernetes, meaning they are scheduled and managed as a single entity. When a pod is created, Kubernetes schedules it to run on a node in the cluster, and all containers within the pod are co-located on the same node.
3. **Shared Resources:** Pods share certain resources, such as networking and storage, among their containers. They can mount shared volumes to exchange data or share configuration files. Additionally, pods share the same IP address and port space, enabling easy communication between containers within the same pod.
4. **Lifecycle:** Pods have a lifecycle that includes creation, running, termination, and deletion. Kubernetes manages pod lifecycles automatically, ensuring that pods are always in the desired state as specified in the deployment manifest.
5. **Abstraction Layer:** Pods serve as an abstraction layer that decouples the application from the underlying infrastructure. They provide a consistent and portable runtime environment for applications, regardless of the underlying node architecture or deployment environment.
6. **Replication and Scaling:** Kubernetes allows you to scale pods horizontally by creating multiple replicas of the same pod. This enables load balancing and high availability for applications. Kubernetes controllers, such as ReplicaSets and Deployments, manage the replication and scaling of pods based on defined criteria.
7. **Networking:** Each pod in Kubernetes is assigned a unique IP address within the cluster. Pods can communicate with each other using this IP address, regardless of the node they are running on. Kubernetes manages networking configurations and ensures that pods can communicate securely and efficiently.

😄 Use cases of Pods:

Pods in Kubernetes serve various use cases across application deployment, scaling, and management. Here are some common scenarios where pods are utilized:

1. **Microservices Architecture:** Pods are often used to deploy individual microservices within a Kubernetes cluster. Each microservice is encapsulated within its pod, allowing teams to independently develop, deploy, and scale their services without impacting other parts of the application.

2. **Stateless Applications:** Pods are suitable for stateless applications that do not store persistent data locally. Stateless applications can scale horizontally by adding or removing pod replicas based on demand, enabling efficient resource utilization and high availability.
3. **Batch Processing and Data Processing:** Pods can be used to run batch processing jobs or data processing tasks within a Kubernetes cluster. Each pod can execute a specific task or computation, and Kubernetes controllers such as Jobs or CronJobs manage the scheduling and execution of these tasks.
4. **Containerized Workloads:** Pods are ideal for running containerized workloads, such as web servers, application servers, databases, message brokers, and background workers. Kubernetes abstracts away the underlying infrastructure, providing a consistent runtime environment for containerized applications.
5. **Development and Testing Environments:** Pods are commonly used in development and testing environments to deploy and test applications before promoting them to production. Developers can create ephemeral pods for testing new features, debugging issues, or running integration tests in isolation.
6. **Continuous Integration and Continuous Deployment (CI/CD):** Pods play a crucial role in CI/CD pipelines by running build and deployment processes within Kubernetes clusters. CI/CD tools, such as Jenkins or GitLab CI, can spin up pods to build application artifacts, run tests, and deploy applications to Kubernetes environments.
7. **Fault Tolerance and High Availability:** Pods can be replicated across multiple nodes in a Kubernetes cluster to achieve fault tolerance and high availability. Kubernetes controllers, such as ReplicaSets or Deployments, manage the lifecycle of pod replicas, ensuring that the desired number of replicas are always running and healthy.
8. **Data Processing Pipelines:** Pods can be used to create data processing pipelines using frameworks like Apache Spark or Apache Flink. Each stage of the pipeline can be represented by a pod, allowing data to be processed, transformed, and analyzed in a distributed manner within the Kubernetes cluster.
9. **Service Mesh:** Pods are part of the service mesh architecture, where they host sidecar containers (e.g., Istio Envoy proxy) alongside the main application container. These sidecar containers enhance network observability, security, and traffic management capabilities within the Kubernetes environment.

This is a comprehensive guide to working with Pods in Kubernetes, along with a walkthrough of setting up a Kubernetes cluster, creating Pods, and configuring Services using NodePort and LoadBalancer types. It covers key concepts such as:

1. **Pods:** Explains the fundamental aspects of Pods, including their role as the smallest deployable unit in Kubernetes, their shared resources, lifecycle, and use cases.
2. **Use Cases of Pods:** Discusses various scenarios where Pods are commonly used, such as deploying microservices, running stateless applications, batch processing, and more.
3. **Setting Up a Kubernetes Cluster:** Provides step-by-step instructions for creating a multi-node Kubernetes cluster using kops, including commands for cluster creation, validation, and deletion.

4. **Working with Pods and Services:** Guides users through creating Pods and Services in Kubernetes, including writing YAML files for Pod and Service definitions, deploying them to the cluster, and accessing the deployed application.

The guide includes detailed commands and explanations for each step, making it accessible for users who are new to Kubernetes or looking to deepen their understanding of Pods and Services in Kubernetes.

It also demonstrates how to configure Services using both NodePort and LoadBalancer types, highlighting the differences between the two approaches and their respective use cases.

Overall, this guide provides a comprehensive overview of working with Pods and Services in Kubernetes and offers practical examples for users to follow along and apply in their own Kubernetes environments.

😄 To begin with the Lab:

1. If you haven't done previous labs then you have to install kubectl and kops on your instance.
2. Now in this lab you are going to make use of multinode cluster. First you have to create it. For that first you are going to run these commands. So, the last command is to delete the cluster. Do not use it until you are done with this lab or other moving forward.
3. You have to use highlighted commands on when your cluster is not validating itself. Then you have to run these two commands. After that again execute validate command then you will see that your cluster has been validated.

   **kops create cluster --name=cloudservicesdemo.in **
   **--state=s3://demo-kops-bucket --zones=ap-southeast-1a,ap-southeast-1b **
   **--node-count=2 --node-size=t3.small --master-size=t3.medium **
   **--dns-zone=cloudservicesdemo.in **
   --node-volume-size=8 --master-volume-size=8
   ---
   kops update cluster --name cloudservicesdemo.in --state=s3://demo-kops-bucket --yes
   ---
   kops validate cluster --state=s3://demo-kops-bucket
   ---
   kops export kubecfg --admin
   export KOPS_STATE_STORE=s3://demo-kops-bucket
   kops export kubecfg --admin --state=s3://demo-kops-bucket cloudservicesdemo.in
   ---
   kops delete cluster --name cloudservicesdemo.in --state=s3://demo-kops-bucket –yes
4. Below you can see that your cluster validation has been failed then you have to use the highlighted command then run validation command again.

```
ubuntu@ip-172-31-25-114:~$ kops validate cluster --state=s3://demo-kops-bucket
Using cluster from kubectl context: cloudservicesdemo.in

Validating cluster cloudservicesdemo.in

Error: validation failed: unexpected error during validation: error listing nodes: Unauthorized
ubuntu@ip-172-31-25-114:~$
```

```
ubuntu@ip-172-31-25-114:~$ kops export kubecfg --admin
export KOPS_STATE_STORE=s3://demo-kops-bucket
kops export kubecfg --admin --state=s3://demo-kops-bucket cloudservicesdemo.in
Using cluster from kubectl context: cloudservicesdemo.in

kOps has set your kubectl context to cloudservicesdemo.in
kOps has set your kubectl context to cloudservicesdemo.in
```

```
ubuntu@ip-172-31-25-114:~$ kops validate cluster --state=s3://demo-kops-bucket
Using cluster from kubectl context: cloudservicesdemo.in

Validating cluster cloudservicesdemo.in

INSTANCE GROUPS
NAME                            ROLE            MACHINETYPE     MIN     MAX     SUBNETS
control-plane-ap-southeast-1a   ControlPlane    t3.medium       1       1       ap-southeast-1a
nodes-ap-southeast-1a           Node            t3.small        1       1       ap-southeast-1a
nodes-ap-southeast-1b           Node            t3.small        1       1       ap-southeast-1b

NODE STATUS
NAME                    ROLE            READY
i-0640d7da51055bbc9     node            True
i-0cd80ed093978eec4     node            True
i-0e7ea4925468c8be7     control-plane   True

Your cluster cloudservicesdemo.in is ready
```

5. If you run the command to get your nodes then you can see your nodes.

```
ubuntu@ip-172-31-25-114:~$ kubectl get nodes
NAME                    STATUS   ROLES           AGE   VERSION
i-0640d7da51055bbc9     Ready    node            11m   v1.26.13
i-0cd80ed093978eec4     Ready    node            11m   v1.26.13
i-0e7ea4925468c8be7     Ready    control-plane   14m   v1.26.13
ubuntu@ip-172-31-25-114:~$
```

6. Now if you run the same command but give your node name and then get output in YAML format. Then you can see the information about your node in YAML format which much easier to read. If you do not specify the output then it will give you the output in JSON format.

   kubectl get node i-0640d7da51055bbc9 -o yaml

```
ubuntu@ip-172-31-25-114:~$ kubectl get node i-0640d7da51055bbc9 -o yaml
apiVersion: v1
kind: Node
metadata:
  annotations:
    csi.volume.kubernetes.io/nodeid: '{"ebs.csi.aws.com":"i-0640d7da51055bbc9"}'
    node.alpha.kubernetes.io/ttl: "0"
    volumes.kubernetes.io/controller-managed-attach-detach: "true"
  creationTimestamp: "2024-04-23T08:24:10Z"
  labels:
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/instance-type: t3.small
    beta.kubernetes.io/os: linux
    failure-domain.beta.kubernetes.io/region: ap-southeast-1
    failure-domain.beta.kubernetes.io/zone: ap-southeast-1a
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: i-0640d7da51055bbc9
    kubernetes.io/os: linux
    node-role.kubernetes.io/node: ""
    node.kubernetes.io/instance-type: t3.small
    topology.ebs.csi.aws.com/zone: ap-southeast-1a
    topology.kubernetes.io/region: ap-southeast-1
    topology.kubernetes.io/zone: ap-southeast-1a
  name: i-0640d7da51055bbc9
  resourceVersion: "3216"
  uid: 5e0a3d8d-5bfa-45da-b7f2-f77bfd0dce68
spec:
  podCIDR: 100.96.1.0/24
  podCIDRs:
```

7. Now to run a pod first you are going to create a directory.
8. Here we have created a directory with name definitions then we have created a folder inside of it with the name pod. After that in pod folder we are going to create our YAML file and write our code inside of it.

```
ubuntu@ip-172-31-25-114:~$ mkdir definitions
ubuntu@ip-172-31-25-114:~$ cd definitions
ubuntu@ip-172-31-25-114:~/definitions$ mkdir pod
ubuntu@ip-172-31-25-114:~/definitions$ cd pod
ubuntu@ip-172-31-25-114:~/definitions/pod$ vim nginxpod.yaml
```

9. Below you can see that we have write our code inside our file nginxpod.yaml then just save and quit.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
```

ports:

- containerPort: 80

```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ cat nginxpod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

10. After that first we are creating our pod. Then we will see that our pod is created or not. After that we can also describe what is happening in our pod.

    kubectl create -f nginxpod.yaml

    kubectl get pod

    kubectl descripe pod nginx

```
ubuntu@ip-172-31-25-114:~/definitions/pod$ kubectl create -f nginxpod.yaml
pod/nginx created
ubuntu@ip-172-31-25-114:~/definitions/pod$ kubectl get pod
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          2m18s
ubuntu@ip-172-31-25-114:~/definitions/pod$
```

11. From above you can see that our pod is running now. If you will run the describe command then you can read the contents in it.

```
ubuntu@ip-172-31-25-114:~/definitions/pod$ kubectl describe pod nginx
Name:             nginx
Namespace:        default
Priority:         0
Service Account:  default
Node:             i-0cd80ed093978eec4/172.20.89.183
Start Time:       Tue, 23 Apr 2024 08:46:30 +0000
Labels:           app=nginx
Annotations:      kubernetes.io/limit-ranger: LimitRanger plugin set: cpu request for container appcontainer
Status:           Running
IP:               100.96.2.240
IPs:
  IP:  100.96.2.240
Containers:
  appcontainer:
    Container ID:   containerd://5b8209766b85b7178b9c52655d14632750b226d36f8d5de51f12acaf4788207d
    Image:          nginx:latest
    Image ID:       docker.io/library/nginx@sha256:0463a96ac74b84a8a1b27f3d1f4ae5d1a70ea823219394e131f5bf3536674419
    Port:           8080/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Tue, 23 Apr 2024 08:46:40 +0000
    Ready:          True
    Restart Count:  0
    Requests:
      cpu:        100m
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-cl69f (ro)
Conditions:
  Type              Status
```

12. Below you can see the events that happened when your pod was created.

```
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  3m35s  default-scheduler  Successfully assigned default/nginx to i-0cd80ed093978eec4
  Normal  Pulling    3m34s  kubelet            Pulling image "nginx:latest"
  Normal  Pulled     3m25s  kubelet            Successfully pulled image "nginx:latest" in 9.051528738s (9.051534881s including waiting)
  Normal  Created    3m25s  kubelet            Created container appcontainer
  Normal  Started    3m25s  kubelet            Started container appcontainer
```

😄 Services: Using NodePort

1. Now we are going to run services on our pod.
2. Below you can see that we have our pod and it is running.

```
ubuntu@ip-172-31-25-114:~/definitions/pod$ cat nginxpod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
    - name: appcontainer
      image: nginx:latest
      ports:
        - name: nginx-port
          containerPort: 8080
ubuntu@ip-172-31-25-114:~/definitions/pod$ kubectl get pod
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          16m
ubuntu@ip-172-31-25-114:~/definitions/pod$
```

3. First thing that you can do is come out to definitions directory then change the name of pod folder to anything you like.
4. Then go back inside of it.

```
ubuntu@ip-172-31-25-114:~/definitions/pod$ cd ..
ubuntu@ip-172-31-25-114:~/definitions$ ls
pod
ubuntu@ip-172-31-25-114:~/definitions$ mv pod demoapp
ubuntu@ip-172-31-25-114:~/definitions$ ls
demoapp
ubuntu@ip-172-31-25-114:~/definitions$ cd demoapp
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ ls
nginxpod.yaml
ubuntu@ip-172-31-25-114:~/definitions/demoapp$
```

5. Now we are going to write the service definition file here.

```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ vim nginx-nodeport.yaml
```

apiVersion: v1
kind: Service
metadata:
name: nginx-service
spec:
type: NodePort
selector:
app: nginx
ports:
- protocol: TCP
port: 80
targetPort: 80
nodePort: 30080 # Choose a port number within the range 30000-32767

```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ cat nginx-nodeport.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30080 # Choose a port number within the range 30000-32767
```

6. After that we are going to create our service file. For that you need to run this command.

   kubectl create -f nginx-nodeport.yaml
   kubectl get svc

7. Below you can see that your service is running as expected. It has its cluster IP and ports mapped on it.

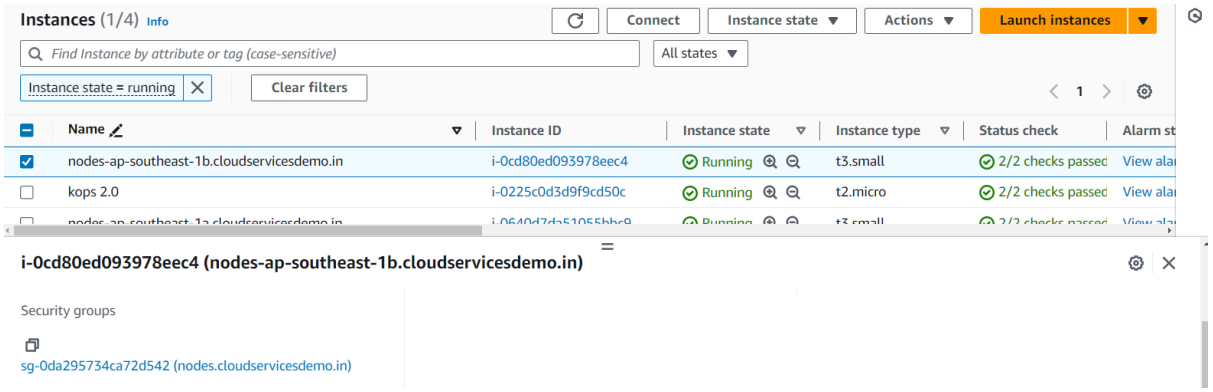
```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ ls
nginxpod-nodeport.yaml  nginxpod.yaml
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl create -f nginxpod-nodeport.yaml
service/demo-service created
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl get svc
NAME           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
demo-service   NodePort    100.64.123.100   <none>        8090:30001/TCP   14s
kubernetes     ClusterIP   100.64.0.1       <none>        443/TCP          46m
ubuntu@ip-172-31-25-114:~/definitions/demoapp$
```

8. Now if you want to see more about this then you can run the describe command.

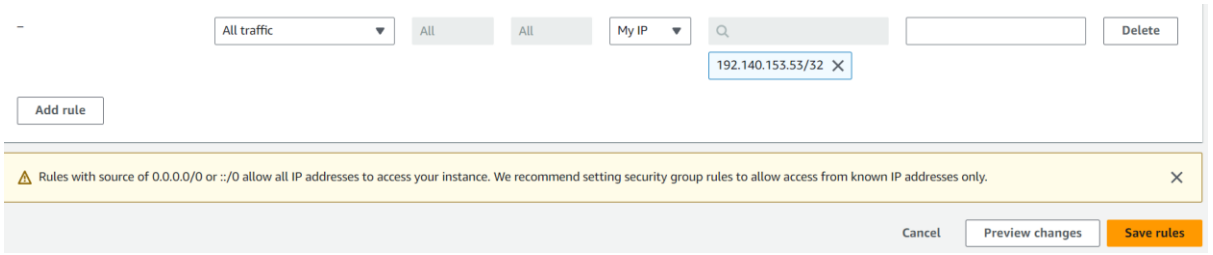
   kubectl describe svc demo-service

```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl describe svc nginx-service
Name:                     nginx-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=nginx
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       100.68.222.149
IPs:                      100.68.222.149
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30080/TCP
Endpoints:                100.96.2.27:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

9. Once you have done all of that now you have to navigate to Console and then go to EC2. There you will see our instances running.
10. From there you have to select any worker node and then go to it security group.



11. In the security group you are going to add one inbound rule which is all traffic from my IP. Because we'll be accessing different ports.



12. Now go to a new tab or new browser copy the public IP address of your instance and paste it there then append 30080 with it.
13. Then you will be able to see your nginx webpage.

😄 Using Load Balancer:

1. Now we are going to use a load balancer which is much more convenient for productions.
2. But first we are going to delete our service definition file. For that run this command

kubectl delete svc nginx-service

```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl get svc
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   100.64.0.1       <none>        443/TCP        3h1m
nginx-service   NodePort    100.68.222.149   <none>        80:30080/TCP   38m
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl delete svc nginx-service
service "nginx-service" deleted
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   100.64.0.1   <none>        443/TCP   3h2m
ubuntu@ip-172-31-25-114:~/definitions/demoapp$
```

3. After that we will create a new file and in that, you are going to paste the below script.

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

4. Create a new file and give it a name then paste the above code in it.
5. After that issue the create command for it.

```
vim nginx-loadbalancer.yml
kubectl create -f nginx-loadbalancer.yml
```

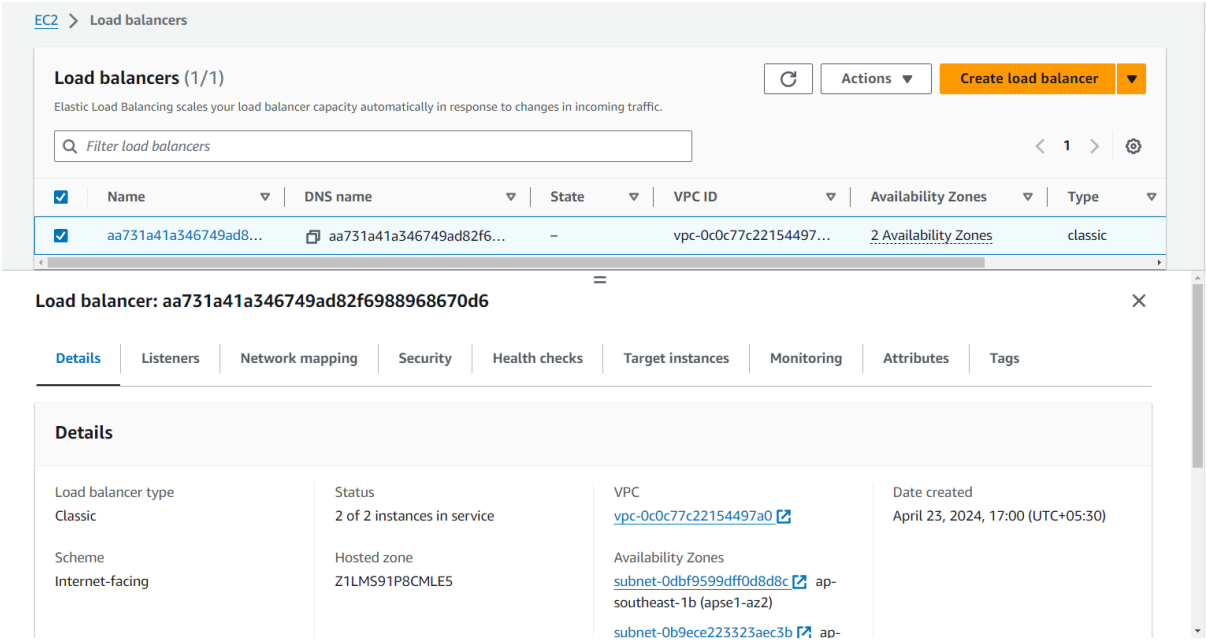
```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ vim nginx-loadbalancer.yml
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl create -f nginx-loadbalancer.yml
service/nginx-service created
ubuntu@ip-172-31-25-114:~/definitions/demoapp$
```

6. Then you can also issue a get command to see your service running. Below you can see the endpoint of the load balancer.

```
kubectl get svc
```

```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl get svc
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP                                                                PORT(S)        AGE
kubernetes      ClusterIP      100.64.0.1     <none>                                                                     443/TCP        3h10m
nginx-service   LoadBalancer   100.65.8.205   aa731a41a346749ad82f6988968670d6-1285447785.ap-southeast-1.elb.amazonaws.com   80:31809/TCP   2m4s
ubuntu@ip-172-31-25-114:~/definitions/demoapp$
```

7. Now if you go to your console and open the load balancer. Below you can see that your load balancer is ready.
8. It takes some time to get the load balancer ready.



9. Once your load balancer is ready then you have to copy the DNS name and paste it in a new browser. Then you will see your nginx web page running.

10. Now if run this command then you can your all services under the current namespace.

```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl get all
NAME            READY   STATUS    RESTARTS   AGE
pod/nginx-pod   1/1     Running   0          59m

NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP                                                              PORT(S)
   AGE
service/kubernetes      ClusterIP      100.64.0.1     <none>                                                                   443/TCP
   3h20m
service/nginx-service   LoadBalancer   100.65.8.205   aa731a41a346749ad82f6988968670d6-1285447785.ap-southeast-1.elb.amazonaws.com   80:31809/TCP
   11m
ubuntu@ip-172-31-25-114:~/definitions/demoapp$
```

11. From here you are going to delete your resources. Below you can see that we have deleted our pod and service.
12. Once we have deleted our service it will also delete our load balancer.

```
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl delete pod/nginx-pod
pod "nginx-pod" deleted
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl delete service/nginx-service
service "nginx-service" deleted
ubuntu@ip-172-31-25-114:~/definitions/demoapp$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   100.64.0.1   <none>        443/TCP   3h28m
ubuntu@ip-172-31-25-114:~/definitions/demoapp$
```

13. You can also delete your cluster if you want.