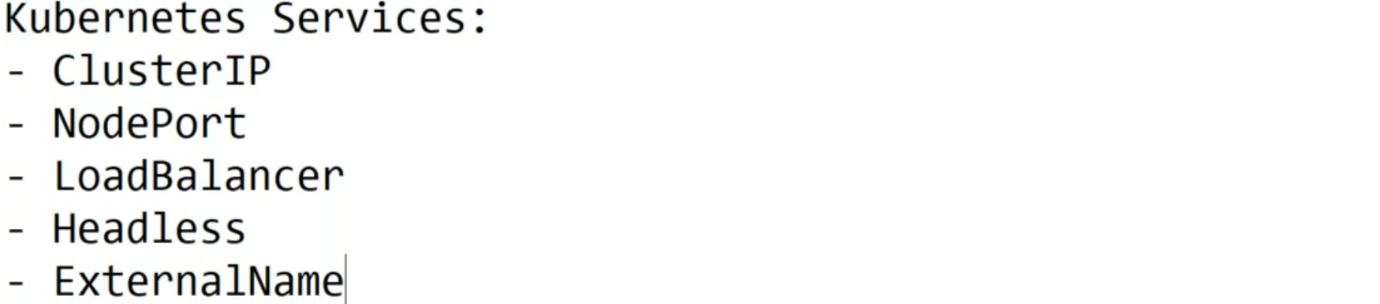
**6. Services**

**Types of services**



**Create Deployment yml file from kubectl command**

--- **kubectl create deployment myapp001 --image=sreeharshav/rollingupdate:v5 --replicas 3 --dry-run=client -o yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

  labels:

    app: myapp001

  name: myapp001

spec:

  replicas: 3

  selector:

    matchLabels:

      app: myapp001

  template:

    metadata:

      labels:

        app: myapp001

    spec:

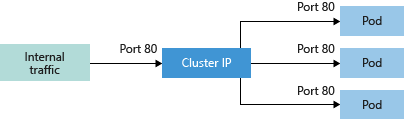
      containers:

      - image: sreeharshav/rollingupdate:v5

        name: rollingupdate

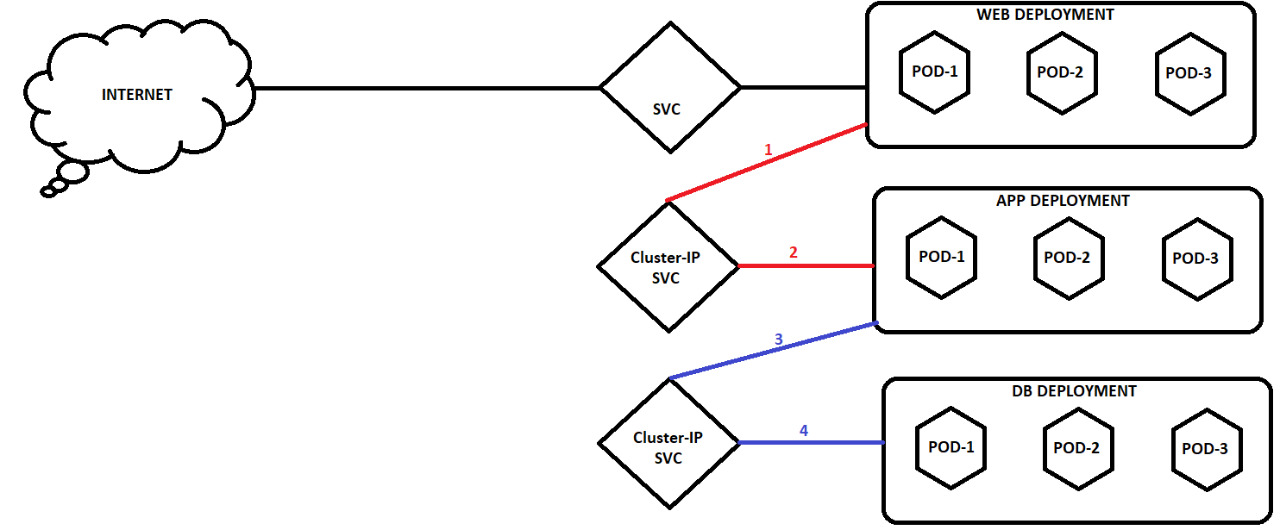
--- **note** – if you use above command then it will generate above command.

**ClusterIP**



--- the clusterIP service is works inside of cluster, if you want to access this clusterIP service from outside of the cluster or from another cluster then it won’t work.

--- **scenario** – when we think of application, we have to think about app servers, web servers and database layers. How we can do these 3 layers in containers.



--- the user first connects to the web deployment, user won’t connect to the app and db deployment’s, so, we don’t have to expose those deployments over internet. The traffic will first come to the web deployment, from there the traffic will goes to the application and db servers.

--- the traffic between app servers and db servers routed through internal traffic. So for these 2 servers, we can use ClusterIP.

**Create ClusterIP yml service file from kubectl command**

--- kubectl expose deployment myapp001 --port=8000 --target-port=80 --dry-run=client -o yaml

apiVersion: v1

kind: Service

metadata:

  creationTimestamp: null

  labels:

    app: myapp001

  name: myapp001

spec:

  ports:

  - port: 8000

    protocol: TCP

    targetPort: 80

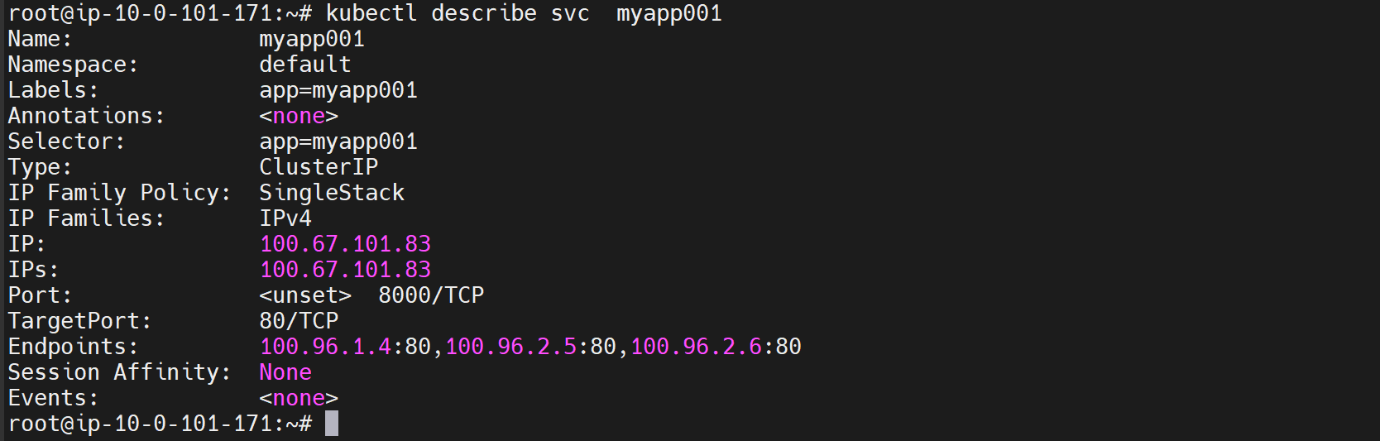
  selector:

    app: myapp001

**test clusterip service, if it is accessible internally**

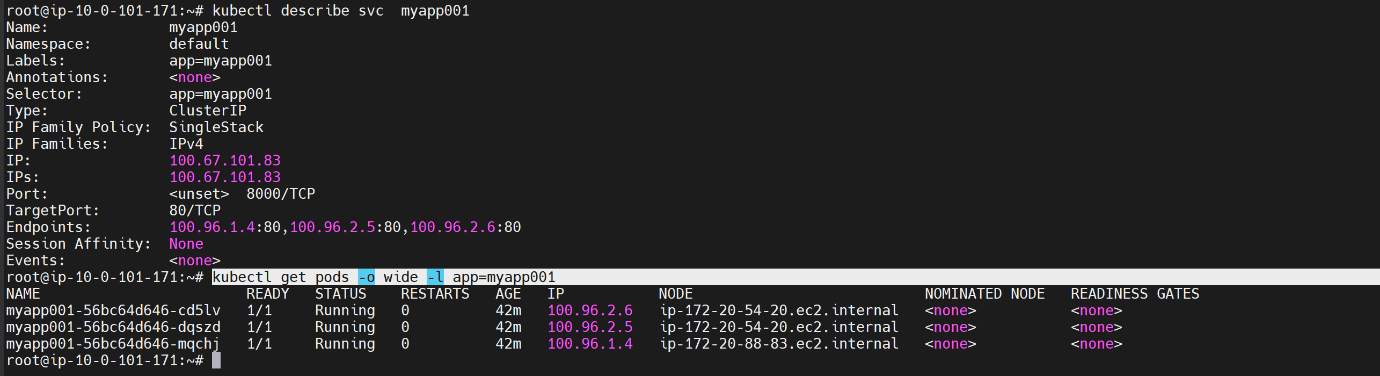
**# List the service**

--- kubectl describe svc myapp001



**# List the pods**

--- kubectl get pods -o wide -l app=myapp001

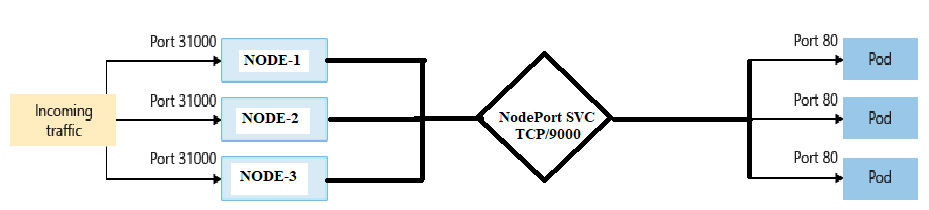


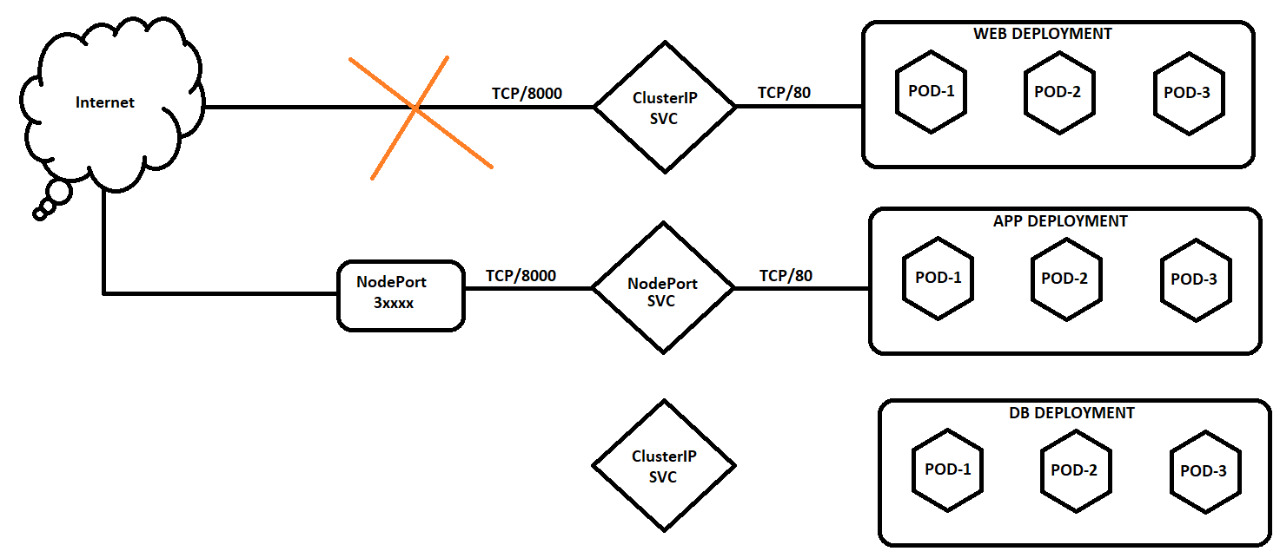
--- **note** – the clusterIP endpoints service IPs should match with pod IP’S. this way you can confirm that the deployment is exposed internally using ClusterIP. Service.

--- **note** – if you hit the clusterip service internally then it will route the traffic to deployment.

**NodePort Service**

--- **NodePort** - Creates a port mapping on the underlying node that allows the application to be accessed directly with the node IP address and port.





**Create NodePort yml service file from kubectl command**

--- **kubectl expose deployment myapp001 --port=8000 --target-port=80 --type=NodePort --dry-run=client -o yaml**

apiVersion: v1

kind: Service

metadata:

  labels:

    app: myapp001

  name: myapp001

spec:

  ports:

  - port: 8000

    protocol: TCP

    targetPort: 80

nodePort:31111

  selector:

    app: myapp001

  type: NodePort

--- **note** – now, you exposed your deployment using NodePort service. Test the deployment by connecting the application over the internet.

**Access the application over the internet**

**# List the nodes**

--- kubectl get nodes -o wide

--- note – copy the external ip address of the nodes.

**# List the service**

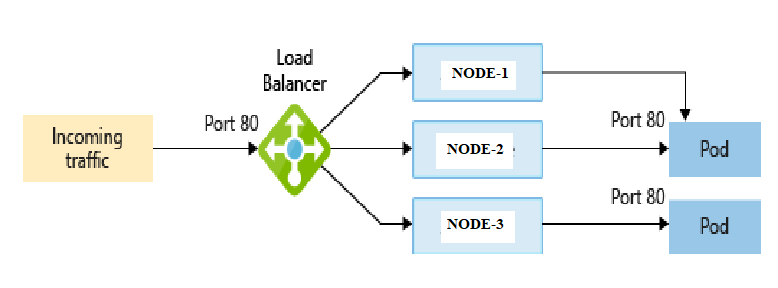
--- kubectl get svc

--- node – copy the node port

**# Access application over internet.**

--- **Error! Hyperlink reference not valid.** IP address>:<service port>

**LoadBalancer**



--- **LoadBalancer** - Creates an Azure/AWS/GCP load balancer resource, configures an external IP address, and connects the requested pods to the load balancer backend pool. To allow customers' traffic to reach the application, load balancing rules are created on the desired ports.

Exposes the Service externally using a cloud provider's load balancer.

Using External Load Balancers per service can become costly and it's not a recommended practise.

The recommended practise is using a Ingress Controller. You can also use [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) to expose your Service. Ingress is not a Service type, but it acts as the entry point for your cluster. It lets you consolidate your routing rules into a single resource as it can expose multiple services under the same IP address

--- **note** – by default, it will create classic load balancer.

**Classic load balancer creating**

--- **kubectl expose deployment myapp001 --port=8000 --target-port=80 --type=LoadBalancer --dry-run=client -o yaml**

apiVersion: v1

kind: Service

metadata:

  labels:

    app: myapp001

  name: myapp001

spec:

  ports:

  - port: 8000

    protocol: TCP

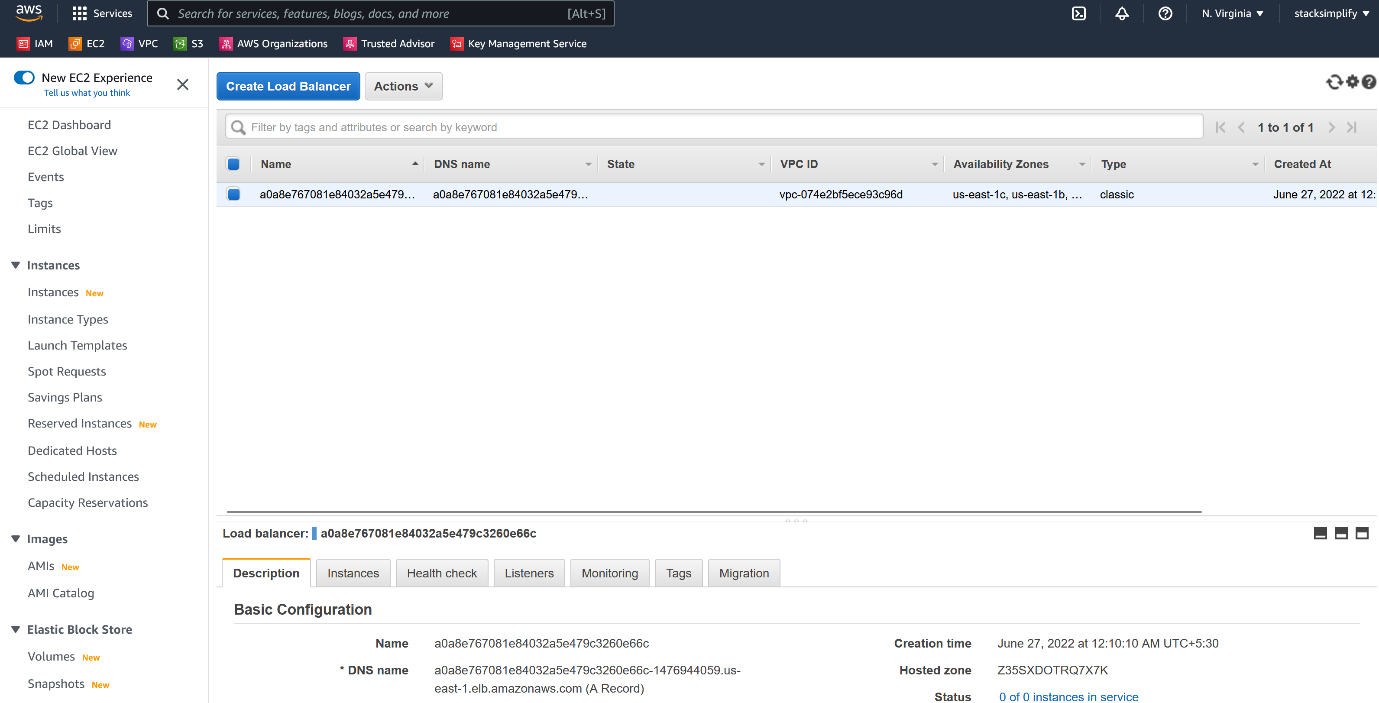
    targetPort: 80

  selector:

    app: myapp001

  type: LoadBalancer

--- **note** – verify the classic load balancer by going to aws.



--- it created classic load balancer.

**Network load balancer**

--- **Annotations Reference** <https://gist.github.com/mgoodness/1a2926f3b02d8e8149c224d25cc57dc1>

--- **note** - if you want to create network load balancer then you have to specify the network load balancer annotation in the service file.

--- **network-loadbalancer.yml**

apiVersion: v1

kind: Service

metadata:

  labels:

    app: myapp001

  name: myapp001

  annotations:

    service.beta.kubernetes.io/aws-load-balancer-type: "nlb"

spec:

  ports:

  - port: 8000

    protocol: TCP

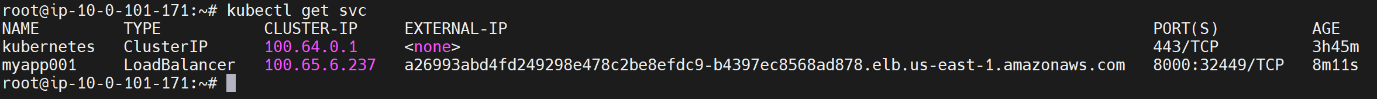
    targetPort: 80

  selector:

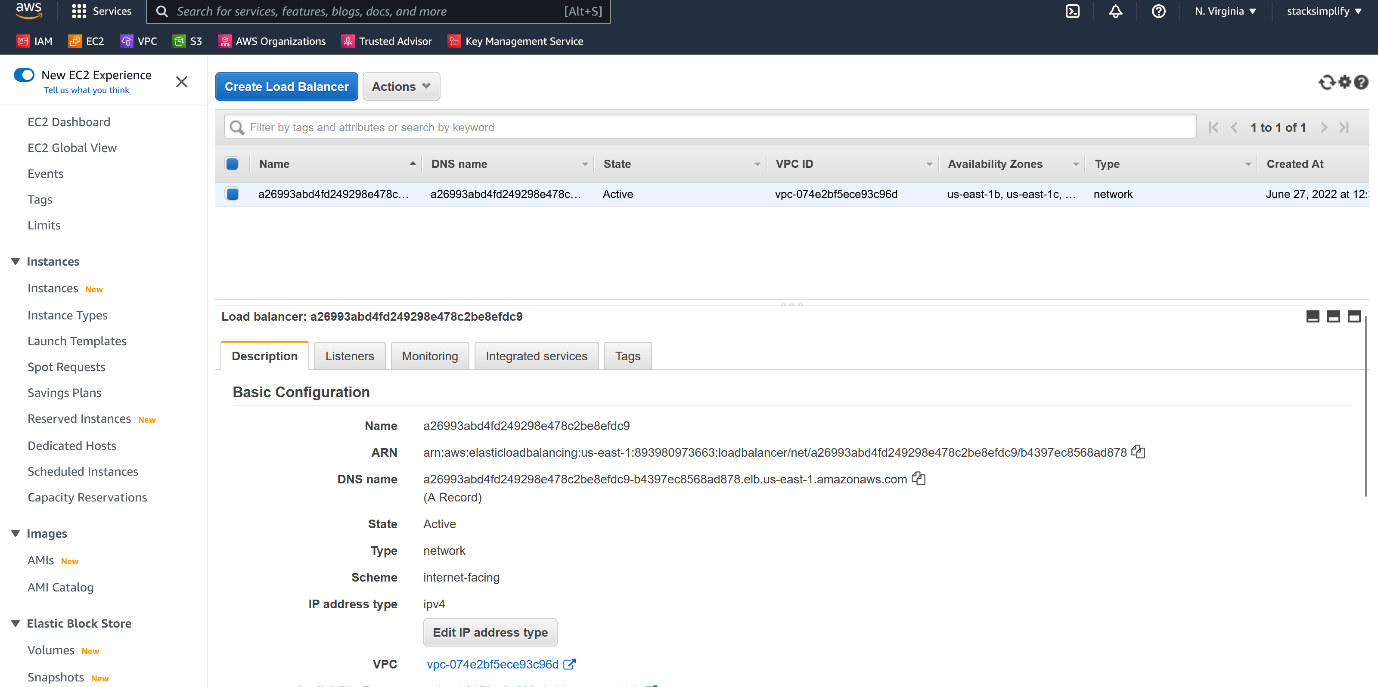
    app: myapp001

  type: LoadBalancer

--- **kubectl get svc**



--- **note** – it created network load balancer, verify the same in aws.



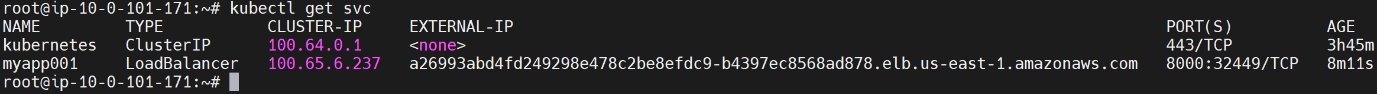
--- it created load balancer type network.

**Access application**

--- **note** – want to access application using load balancer endpoint.

# List the service

--- kubectl get svc

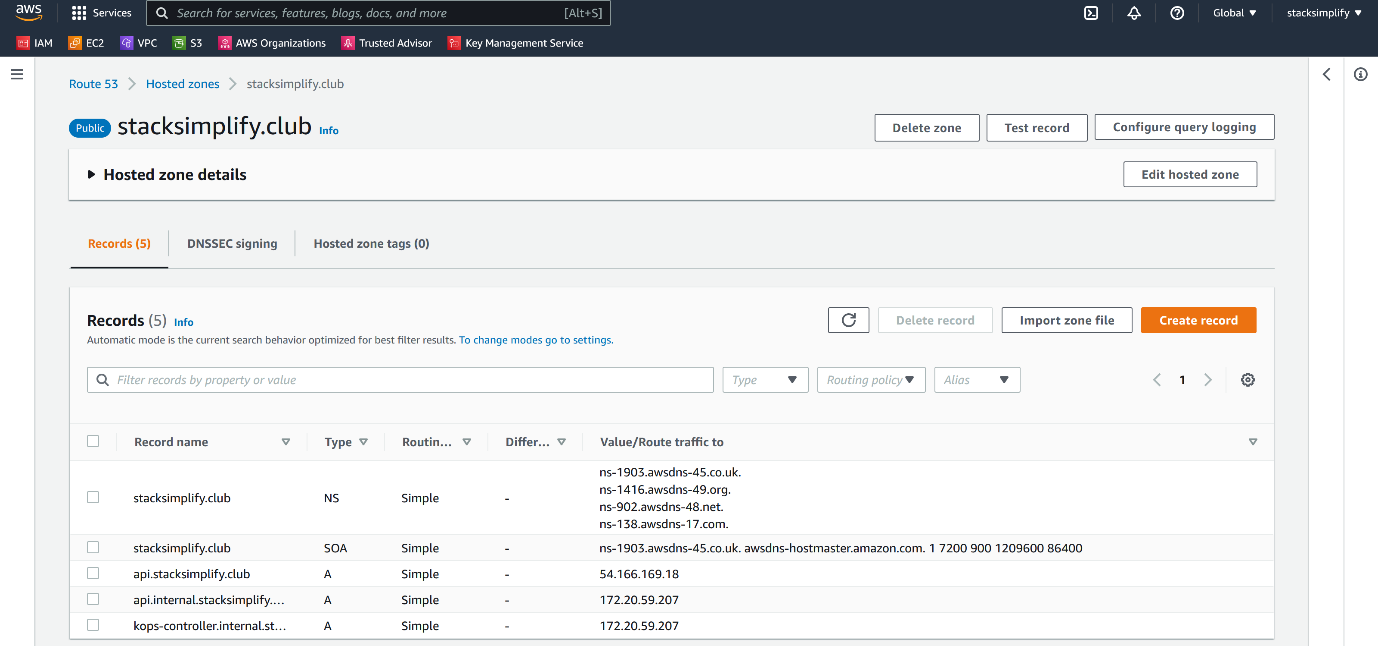


--- copy the load balancer dns name and paste it on web browser with service port, in my case service port is 8000.

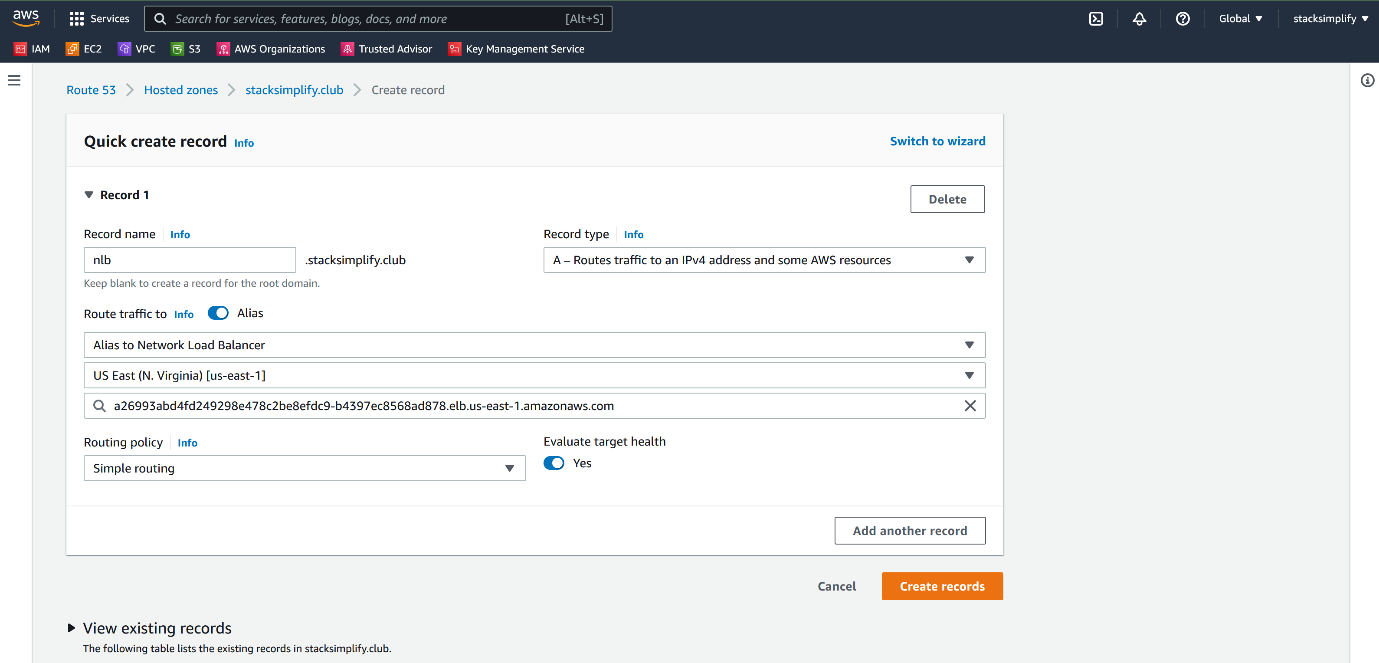
--- **a26993abd4fd249298e478c2be8efdc9-b4397ec8568ad878.elb.us-east-1.amazonaws.com:8000**

**Created record in route53**

--- note – you have a load balancer dns name and you are tasked to route the traffic form route53 subdomain name to load balancer dns name.



--- click on create record.



--- fill the above details and click on create record.

--- **nlb.stacksimplify.club** – type this on web browser and the traffic will go to the load balancer and you will be able to access the application.

**EXTERNALNAME SERVICE**

--- **ExternalName** - Creates a specific DNS entry for easier application access. Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

apiVersion: v1

kind: Service

metadata:

name: google-svc

spec:

type: ExternalName

externalName: www.google.com

ports:

- port: 80

name: http

targetPort: 80

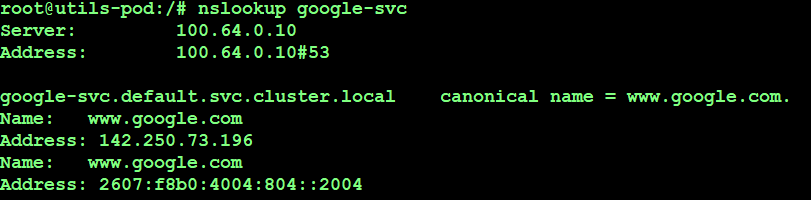
protocol: TCP

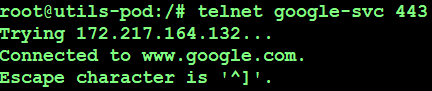
- port: 443

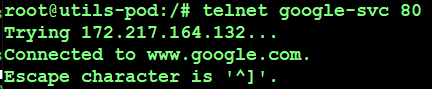
name: htts

targetPort: 443

protocol: TCP







**SERVICE FOR EXTERNAL IP ADDRESS**

--- note – your service will connect to the external service.

---

apiVersion: v1

kind: Service

metadata:

name: mysql

spec:

clusterIP: None

selector:

app: mysql

ports:

- name: mysql

protocol: TCP

port: 3306

targetPort: 3306

---

kind: Endpoints

apiVersion: v1

metadata:

name: mysql

subsets:

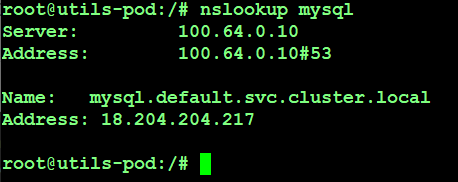
- addresses:

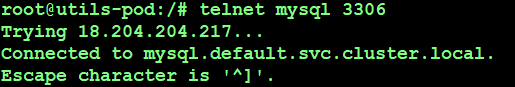
- ip: 18.204.204.217

ports:

- port: 3306

name: mysql





**HEADLESS ADDRESS**

Sometimes you don't need load-balancing and a single Service IP. In this case, you can create what are termed "headless" Services, by explicitly specifying "None" for the cluster IP (.spec.clusterIP).

Headless Service in general used for stateful sets in Kubernetes.

--- **note** – the service route the traffic to the pods.

---

#Headless Service

apiVersion: v1

kind: Service

metadata:

labels:

run: deploy01

name: deploy01

spec:

clusterIP: None

ports:

- port: 80

protocol: TCP

targetPort: 80

selector:

run: deploy01

--- kubectl get svc

