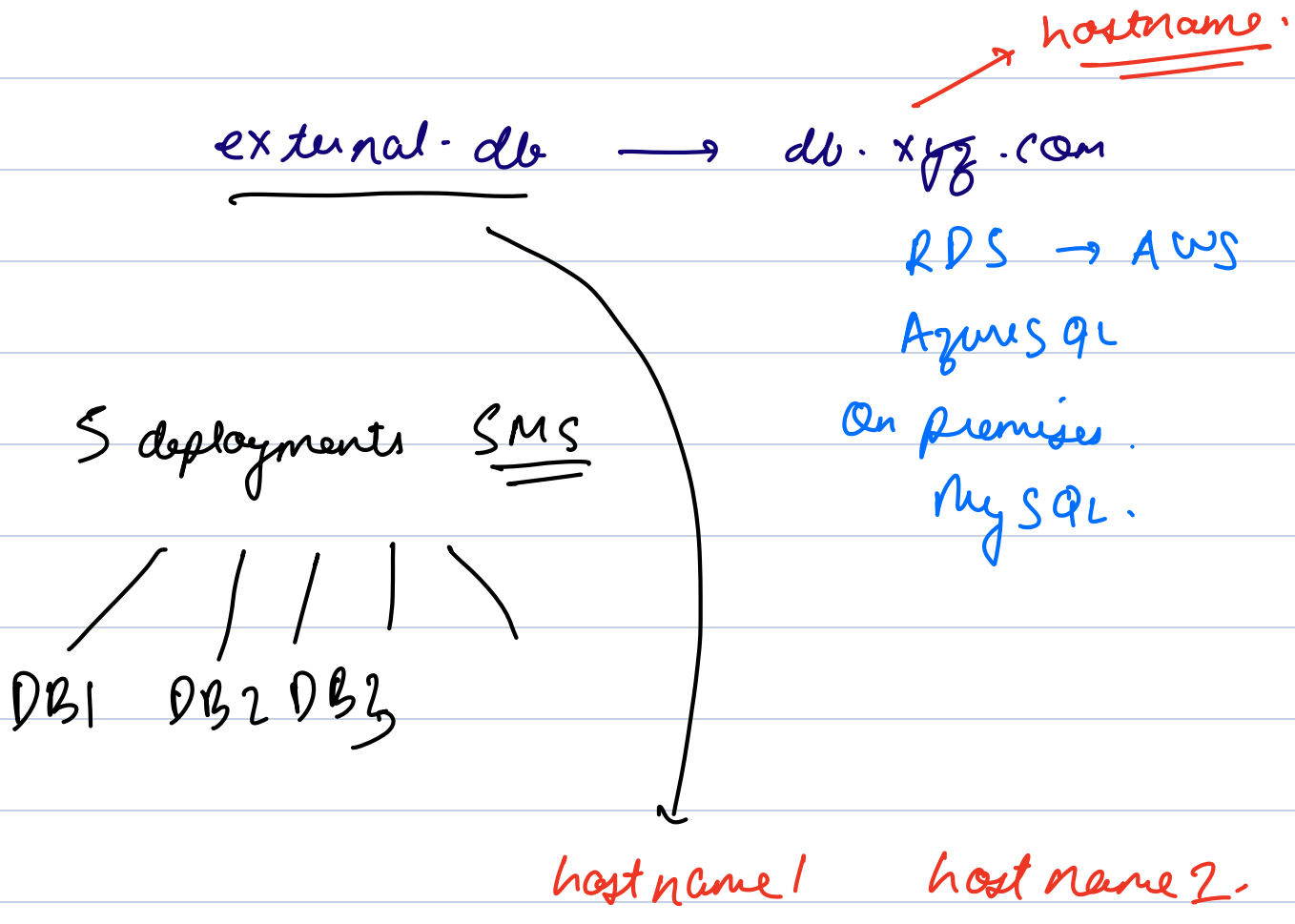
Kubernetes Networking Continued

starts at 9:05 pm

AGENDA

1. External Name Service
2. Headless Service
3. Ingress
4. Network Policies

→ External Name Service



```
apiVersion: v1
kind: Service
metadata:
  name: external-db
spec:
  type: ExternalName
  externalName: example.com  # Redirects traffic to example.com
```

```
kubectl run test-pod --image=busybox --restart=Never -it --rm -- /bin/sh
```

```
nslookup external-db
```

→ Headless service

No Cluster IP No load Balancing.

returns pod IPs.

→ stateful applications (DBs)

The **absence of an actual ClusterIP** tells Kubernetes **not to perform load balancing**

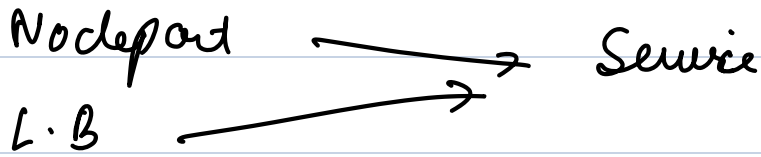
```
apiVersion: v1
kind: Service
metadata:
  name: nginx-headless
spec:
  selector:
    app: nginx
  clusterIP: None  # This makes it a Headless Service
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

nslookup nginx-headless.

Best Practices for using services.

1. Use selectors appropriately.
2. Resource Management.
3. Security.

Ingress.



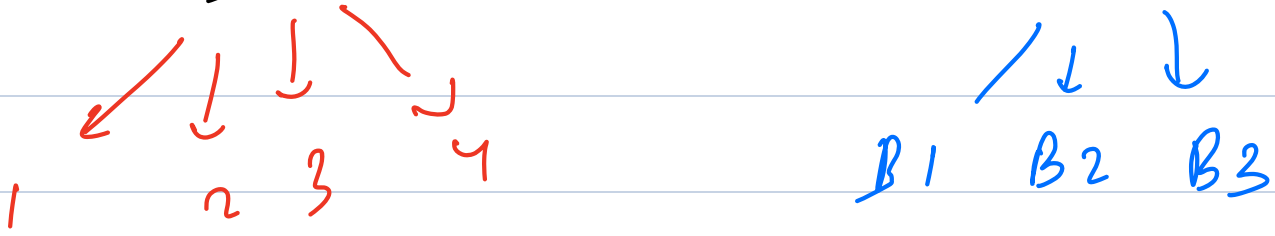
→ Rules.

① hostname based
② Path - Based.

app.example.com → Service 1
api.example.com → service 2

/app1 → service 1
/app 2 → service 2.

amazon.com/Sports → S1
amazon.com/Books → S2



→ Set of Rules.

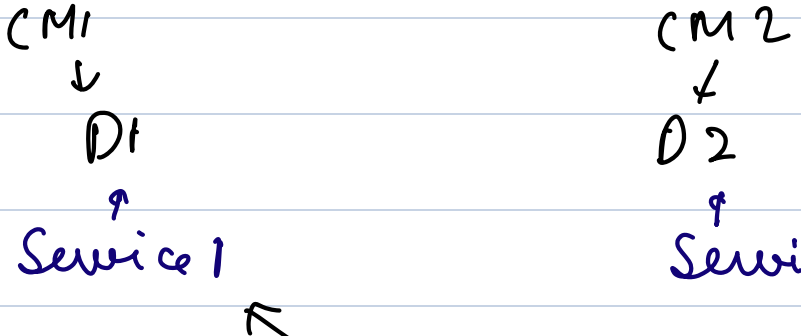
Ingress Controller → implements the ingress rules

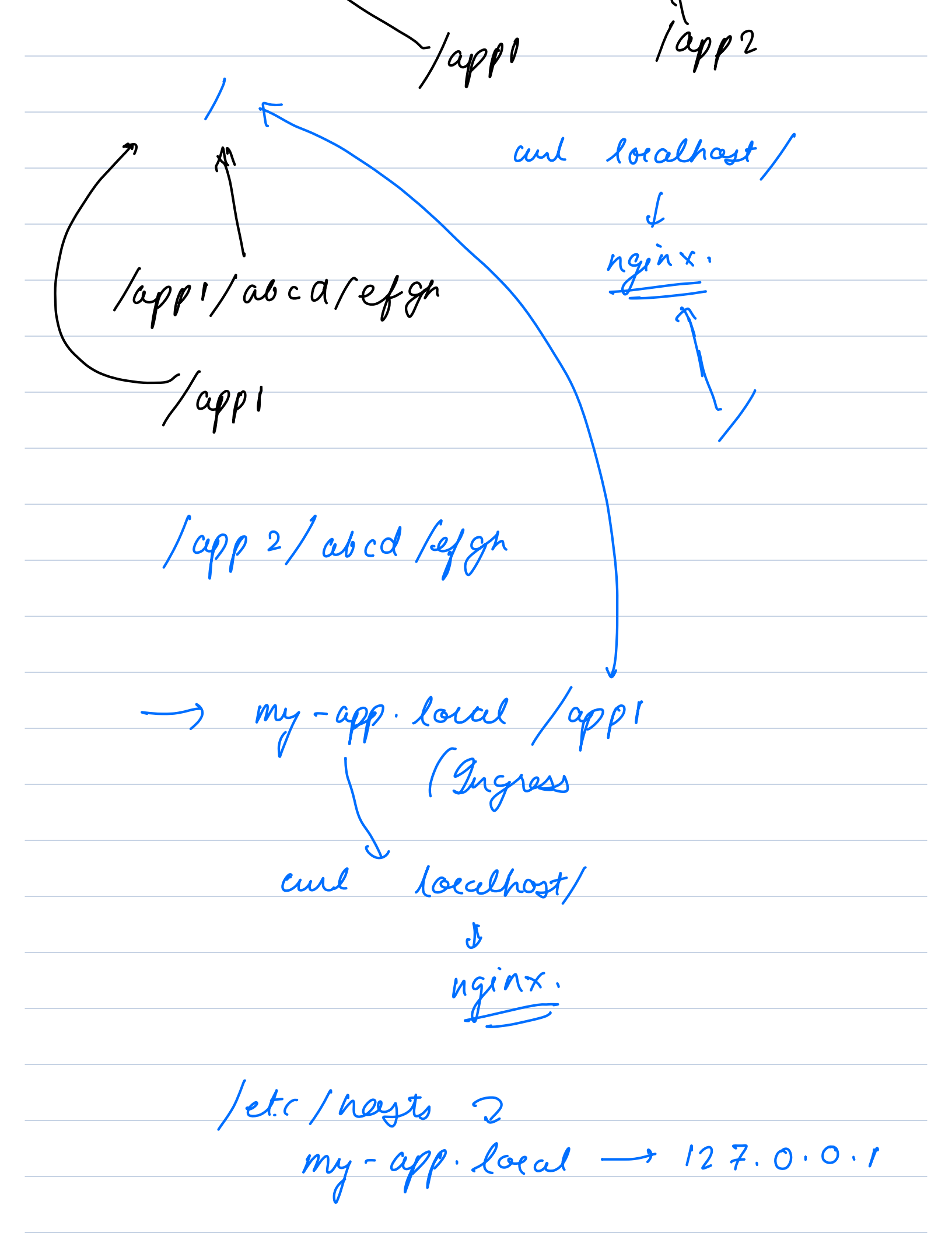
Think of **Ingress** as a blueprint for traffic routing and the **Ingress Controller** as the worker that follows that blueprint to direct traffic properly.

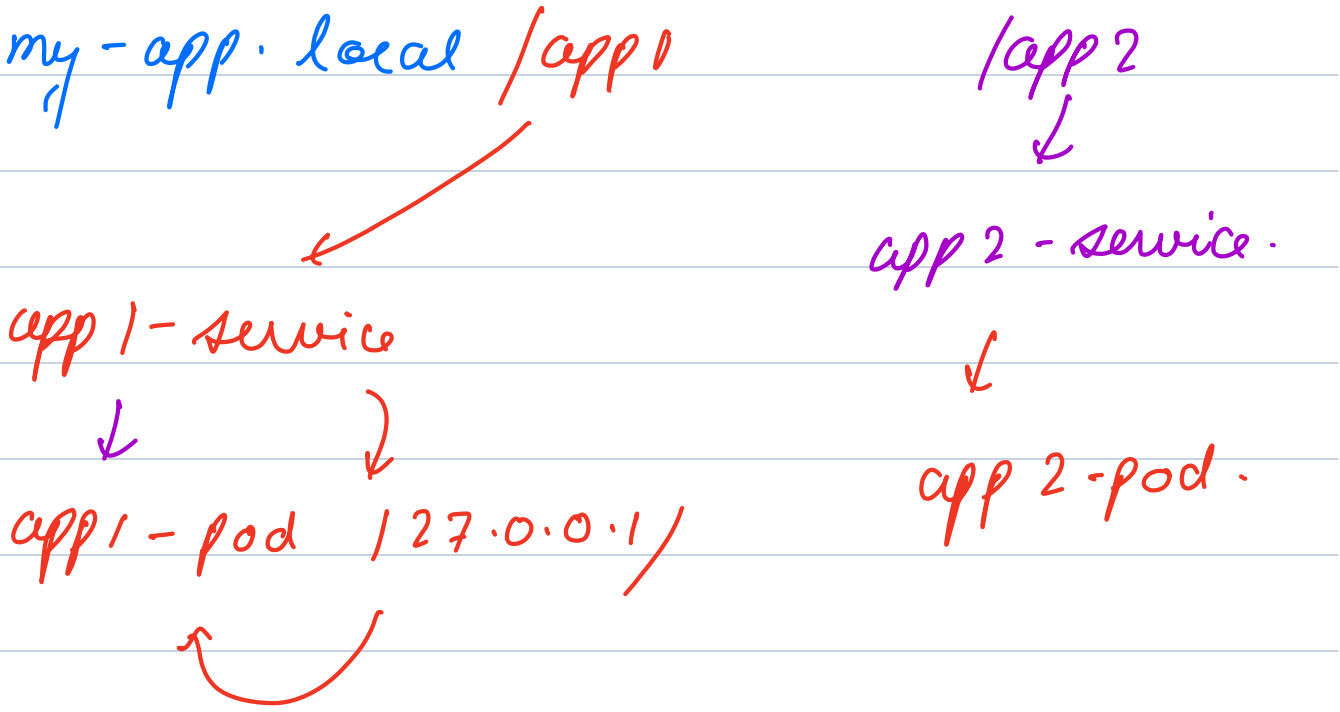
→ Demo.

1. Create a new kind Cluster
2. install nginx ingress controller
3. create deployments and svc.
4. Create ingress Resource
5. Test.

localhost:80 → control-plane:80




/app1
/app2
/
curl localhost/
nginx.
/app1/abcd/efgh
/app1
/app2/abcd/efgh
my-app.local /app1
Ingress
curl localhost/
nginx.
/etc/hosts
my-app.local → 127.0.0.1



1. **`curl my-app.local/app1`** sends a request. → EC2
2. **`/etc/hosts`** maps `my-app.local` to the IP of the **Kind node**. → Control Plane node :80
3. The request reaches the **Ingress Controller** inside the cluster.
4. The Ingress Controller matches the **host** (`my-app.local`) and **path** (`/app1`) in the rule.
5. It routes the request to the **app1-service**.
6. **App1** processes the request and responds.
7. The response returns to **curl**

↓
VM:80
Node:80

→ 10:35 Break.

→ curl localhost/app1 ?

curl localhost/

Ingress.

http://my-app.local/app1 → app1-servi

http://my-app.local/

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/kind/deploy.yaml → Install ingress Controller

kubectl get pods -n ingress-nginx

Create deployment, CM, Service

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app1-config
data:
  index.html: |
```

```
    <html>
    <head><title>App 1</title></head>
    <body style="background-color: lightblue;">
      <h1>Welcome to App 1!</h1>
    </body>
    </html>
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: app2-config
data:
  index.html: |
    <html>
    <head><title>App 2</title></head>
    <body style="background-color: lightgreen;">
      <h1>Welcome to App 2!</h1>
    </body>
    </html>
---
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: app1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app1
  template:
    metadata:
      labels:
        app: app1
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config-volume
          mountPath: /usr/share/nginx/html
      volumes:
      - name: config-volume
```

```
      configMap:
        name: app1-config
---
apiVersion: v1
kind: Service
metadata:
  name: app1-service
spec:
  selector:
    app: app1
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2
spec:
  replicas: 1
  selector:
```

```
  matchLabels:
    app: app2
  template:
    metadata:
      labels:
        app: app2
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config-volume
          mountPath: /usr/share/nginx/html
      volumes:
      - name: config-volume
        configMap:
          name: app2-config
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: app2-service
spec:
  selector:
    app: app2
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80


apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: my-app.local
    http:
      paths:
```

create ingress Resource

```
    - path: /app1
      pathType: Prefix
      backend:
        service:
          name: app1-service
          port:
            number: 80
    - path: /app2
      pathType: Prefix
      backend:
        service:
          name: app2-service
          port:
            number: 80
```

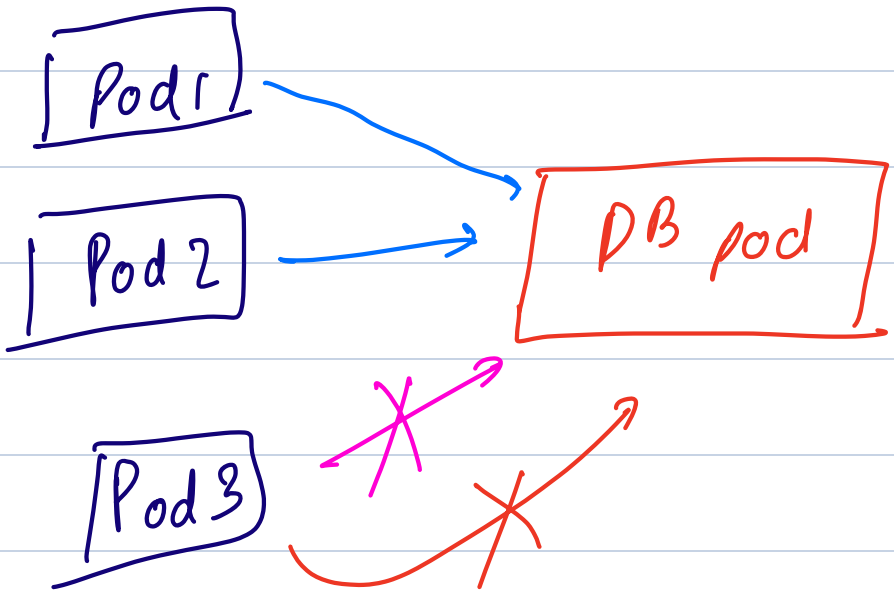
```
sudo vi /etc/hosts
```

```
127.0.0.1 my-app.local
```

```
curl http://my-app.local/app1
curl http://my-app.local/app2
```

Network Policies

Network Policies in Kubernetes define **rules for controlling** how Pods communicate with each other and external resources.



Key Concepts of Network Policies

1. **Pod Selector**: A label selector to select which pods the policy applies to. app = nginx.

2. **Ingress and Egress Rules**:

   - **Ingress**: Defines the allowed incoming traffic to a pod.

   - **Egress**: Defines the allowed outgoing traffic from a pod.

3. **Namespace Selector**: Used to specify namespaces from which traffic is allowed.

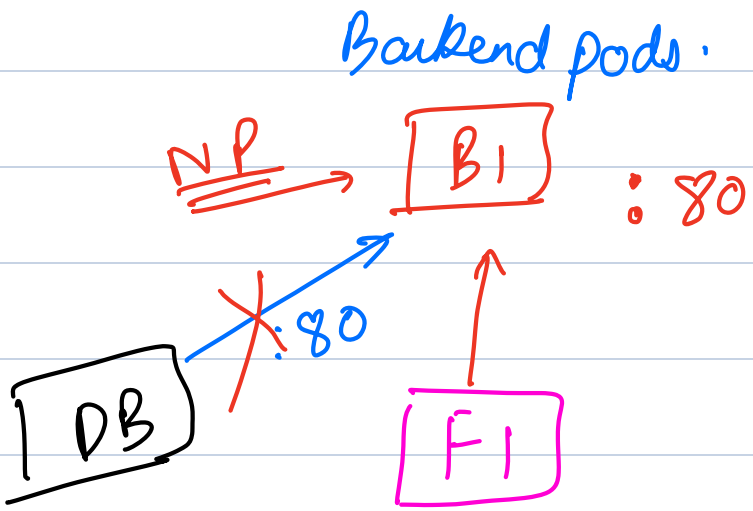
4. **IP Blocks**: Allows you to specify traffic from specific IP ranges.

5. **Ports**: You can restrict traffic based on ports.

Ingress → Incoming traffic pod / svc

Egress → Outgoing traffic

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend
      ports:
        - protocol: TCP
```



port: 80

→ Container Network Interface.

Kindnet → Network Policies.

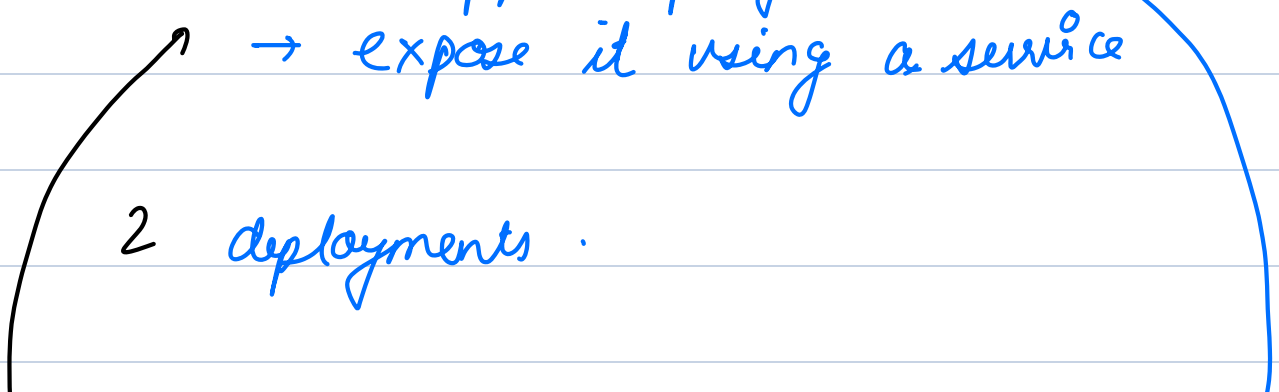
Feature	CNI (Container Network Interface)	kube-proxy
Primary Role	Manages pod networking (IP allocation & routing)	Manages Service networking (routing & load balancing)
Scope	Handles Pod-to-Pod communication	Handles Service-to-Pod communication
How It Works	Sets up network interfaces, assigns IPs, configures routes	Uses iptables/ipvs to route traffic to the correct pod
Used For	Pod networking, enforcing network policies (if supported)	Service discovery and load balancing
Examples	Flannel, Calico, Cilium, Weave	Built-in Kubernetes component (`kube-proxy`)
Dependency	Required for Kubernetes networking	Optional with advanced CNIs (e.g., Cilium can replace `kube-proxy`)
Load Balancing	Not responsible for load balancing	Provides basic load balancing for services
Handles Network Policies?	Some CNIs (Calico, Cilium) support them	Does not handle network policies

→ Created NS

→ web app deployment

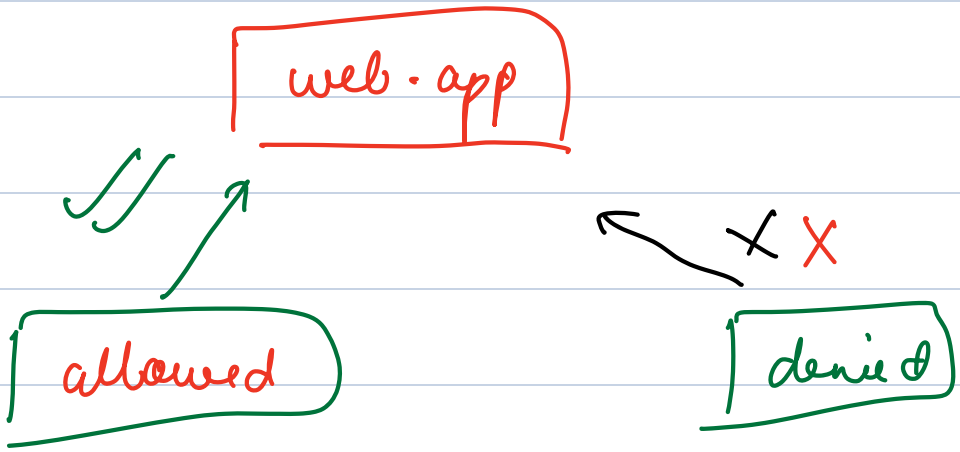
→ expose it using a service

2 deployments.



allowed

denied




```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  disableDefaultCNI: true  # Disables Kindnet
  podSubnet: "192.168.0.0/16"  # Ensure compatibility with Calico
nodes:
- role: control-plane
- role: worker
```

```
kind create cluster --config kind-calico.yaml --name
```

Install Calico

kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/calico.yaml

We will create a NAMESPACE.

kubectl create namespace network-policy-lab

Creating Web Application Deployment

kubectl create deployment web-app --image=nginx -n network-policy-lab

kubectl expose deployment web-app --port=80 --target-port=80 -n network-policy-lab

Verify the deployment and Service

kubectl get deployments -n network-policy-lab

kubectl get services -n network-policy-lab

Deploy 2 pods using deployment which we will try to connect to web application.

kubectl create deployment busybox-allowed --image=busybox -n network-policy-lab -- /bin/sh -c "sleep 3600"

kubectl create deployment busybox-denied --image=busybox -n network-policy-lab -- /bin/sh -c "sleep 3600"

Verify the deployments

kubectl get deployments -n network-policy-lab

Without Creating any Network Policies lets try to communicate with the web application

kubectl exec -it podname --sh

wget -qO- http://web-app

Able to communicate using both the pods.

Now we create a Network Policy

It controls **Ingress traffic**, meaning it specifies which pods are **allowed** to send traffic to the selected pods.

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: allow-busybox-allowed

  namespace: network-policy-lab

spec:

```
  podSelector:
    matchLabels:
      app: web-app
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: busybox-allowed


apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
  namespace: network-policy-lab
spec:
  podSelector:
    matchLabels:
      app: web-app
  policyTypes:
- Ingress
```