**Agenda: Services and Ingress**

1. Kubernetes Services
2. Service Types
   * ClusterIP Service
   * NodePort Service
   * LoadBalancer Service
   * External Service
3. Kubernetes Services DNS
4. Ingress Controllers and Alternatives
5. Setting up Ingress Locally

* Creating the Ingress Config

Kubernetes Services

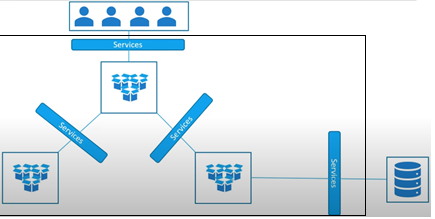
**The Problem:**

* In Kubernetes, if you use a Deployment to run your app, it can create and destroy Pods dynamically. Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.
* This can lead to a problem: if some set of Pods (call them “backends”) provides functionality to other Pods (call them “frontends”) inside our cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

This is where **Services** can be helpful.

**Service Resources:**

* Kubenetes Service is an abstract way to **expose** an application(Microservice) running on a **logical set of Pods** and a **policy** by which to access them.
* Services enable a **loose coupling** between dependent Pods. Frontend Pods don’t need to know the direct IP address of the Backend Pods.



* A **Service** in Kubernetes is a **REST object**, similar to a Pod.
* Like all of the REST objects, you can POST a Service definition to the **API server** to create a new instance.
* The name of a Service object must be a valid **DNS label name**.

**ServiceTypes:**

* Pod IP address are not exposed outside the cluster without a Service. Services allow our applications to receive traffic from outside the cluster.
* For some parts of our application (for example, frontends) we may want to expose a Service on to an external IP address, i.e. outside of our cluster.
* Kubernetes ServiceTypes allow us to specify what kind of Service we want.

1. ClusterIP
2. LoadBalancer
3. NodePort
4. ExternalName

A blue dot in a rectangle

Description automatically generated

**CLUSTER-IP SERVICE**

* Exposes the Service on a **cluster-internal IP**. Choosing this value makes the Service **only reachable** from within the cluster.
* This is the **default** ServiceType.
* ClusterIP is the preferred option for internal/backend service access and uses an internal/private IP address to access the service.
* Some examples of where ClusterIP might be the best option include service debugging during development and testing and internal traffic.

A diagram of a cluster ip

Description automatically generated

**Steps to create a ClusterIP Service:**

1. kubectl **create** deployment mynginx-deployment --image nginx
2. kubectl scale deploy/mynginx-deployment --replicas 5
3. kubectl **expose** deploy/mynginx-deployment --name my-service-cip **--type=ClusterIP** --port 8080 --target-port=80

Note: Expose uses service creating generator to create the service.

If the service name is not mentioned, it uses the name of deployment. In this case, it would be **mynginx**

1. **kubectl get all** OR **kubectl get service**

Note the ClusterIP for further use (eg: 10.99.158.232)

OR

**Deployment.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

spec:

  replicas: 10

  selector:

    matchLabels:

      app: **nginx-app**

  template:

    metadata:

      labels:

        app: nginx-app

    spec:

      containers:

      - name: mynginx-container

        image: nginx

        ports:

        - containerPort: 80

**Service.yaml**

apiVersion: v1

kind: Service

metadata:

  name: my-service-cip

spec:

  type: ClusterIP

  selector:

    app: nginx-app

  ports:

    - protocol: TCP

      port: 8080

      targetPort: 80

1. kubectl apply -f service.yaml

* This specification creates a new Service object named “**my-service-cip**”, which targets TCP port **80** on any Pod with the **app=nginx-app** label.
* Kubernetes assigns this Service an IP address (sometimes called the "cluster IP").
* The controller for the Service selector continuously scans for Pods that match its selector, and then POSTs any updates to an **Endpoint object** also named “my-service”.

A picture containing diagram, line, sketch, technical drawing

Description automatically generated

**Testing if the service is forwarding the traffic to Pods**

1. kubectl get services
2. kubectl get pods -o wide
3. kubectl get endpoints
4. kubectl get endpoints -oyaml
5. kubectl describe endpoints my-service-cip #Notice that it lists IP addresses of all Pods
6. kubectl **run** myubuntu -it --rm --image=ubuntu -- sh

# apt update

# apt upgrade

# apt install curl -Y

# curl [http://**my-service-cip**:8080](http://my-service-cip:8080)

# exit

1. **kubectl run -it --rm mypod --image=nginx --restart=Never -- curl http://**my-service-cip**:8080**

**kubectl run -it --rm mypod --image=nginx --restart=Never -- curl http://**10.99.158.232**:8080**

**We can use nslookup to get the fully qualified domain name (FQDN) of the service.**

**To install nslookup:**

* apt update
* apt install dnsutils
* **nslookup my-service-cip** #This will show the FQDN
* curl <http://my-service-cip.default.svc.cluster.local:8080> #[default] is the namespace in the url

**To access the Service from a different namespace. It is mandatory to use the Fully Qualifed Name (FQN) of a service.**

1. kubectl create ns demo-namespace
2. kubectl **run** mynginx -it --rm --image=nginx **-n demo-namespace** -- sh

# curl [http://**my-service-cip**:8080](http://my-service-cip:8080) #Fail

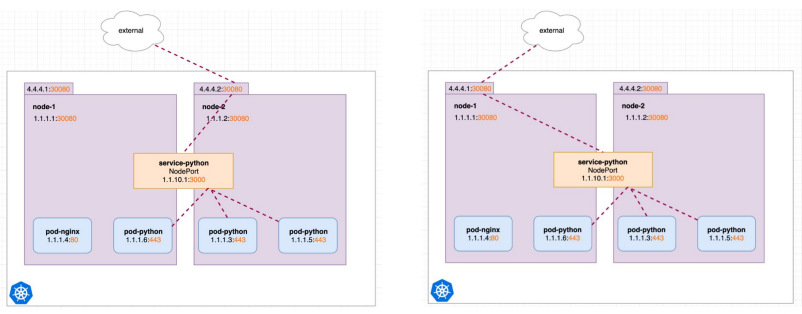
# curl [http://**my-service-cip.default.svc.cluster.local**:8080](http://my-service-cip.default.svc.cluster.local:8080)

# curl [http://**my-service-cip.default**:8080](http://my-service-cip.default:8080)

# exit

**NODEPORT SERVICE**:

* Exposes the Service on each Node’s IP at a static port (the NodePort [**Hi Range Port**]). High range port (30000 to 32767) used for exposing NodePort.
* Each node proxies that port (the same port number on every Node) into your Service
* A ClusterIP Service, to which the NodePort Service routes, is automatically created.
* Useful when front end pods are to be exposed outside the cluster for users to access it.
* We will be able to contact the NodePort Service, **from outside the cluster**, by requesting **<NodeIP>:<NodePort>.**
* Same NodePort will be used across all the nodes in a multi node cluster. Also, same service (single instance) can be used to access container in any Pod or any Node in the cluster.



**Create NodePort type of Service:**

1. **kubectl expose deployment nginx-deployment --target-port=80 --port=8080 --type=NodePort --name=my-service-np --selector="app=nginx-deployment"**

Note that 8080 is port for NodePort which is 30008.

1. kubectl get service

Note the High Port under PORT(S) section (Eg: 30008)

1. **For Docker Desktop**

curl http://localhost:<high-port> or open in web browser

**OR**

**For Minikube:**

minikube ip

curl http://<minikube-ip>:30008

We can access the service through localhost without getting into the cluster. This works only in Docker and not in minicube.

1. kubectl get svc **my-service-np** -o yaml

Note: 3 ports are involved

1. 30008 (High Port) is of NodePort Service
2. 8080 is of ClusterIP
3. 80 is Application port in container

|  |  |
| --- | --- |
| apiVersion: v1  kind: Service  metadata:    name: my-service-np  spec:    type: NodePort    selector:      app: nginx-app    ports:    - protocol: TCP      nodePort: 30008 #Port of Node      port: 8080 #Port of Service      targetPort: 80 #Port of Container |  |

**LOAD BALANCER SERVICE**

Exposes the Service externally using a **cloud provider’s load balancer**. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created and are hidden from us. It assigns a fixed external IP to the Service. It is superset of NodePort.

There is no filtering, no routing, etc. This means you can send almost any kind of traffic to it, like HTTP, TCP, UDP or WebSocket's

This works at **Layer4** of OSI layers.

Diagram

Description automatically generated

**Create LoadBalancer type of Service:**

1. kubectl expose deploy/mynginx --port 8080 --target-port=80 --type=LoadBalancer --name my-service-lb
2. **curl localhost:port** or **open in web browser**

We can access the service through localhost without getting into the cluster.

apiVersion: v1

kind: Service

metadata:

  name: my-service-lb

spec:

  type: LoadBalancer

~~loadBalancerIP: 10.10.10.10~~

  selector:

    app: nginx-app

  ports:

    - name: http

      protocol: TCP

      port: 8080

      targetPort: 80

Some cloud providers allow you to specify the **loadBalancerIP**. In those cases, the load-balancer is created with the user-specified loadBalancerIP. If the loadBalancerIP field is not specified, the loadBalancer is set up with an ephemeral IP address. If you specify a loadBalancerIP but your cloud provider does not support the feature, the loadbalancerIP field that you set is ignored.

**Note:**

On **Azure**, if you want to use a user-specified public type **loadBalancerIP**, you first need to create a **static** type public IP address resource. This public IP address resource should be in the same resource group of the other automatically created resources of the cluster. For example, **MC\_myResourceGroup\_myAKSCluster\_eastus**.

Specify the assigned IP address as **loadBalancerIP**. Ensure that you have updated the securityGroupName in the cloud provider configuration file.

For information about troubleshooting CreatingLoadBalancerFailed permission issues see, [Use a static IP address with the Azure Kubernetes Service (AKS) load balancer](https://docs.microsoft.com/en-us/azure/aks/static-ip) or [CreatingLoadBalancerFailed on AKS cluster with advanced networking](https://github.com/Azure/AKS/issues/357).

**Lmitations with LoadBalancer:**

* Every service exposed will gets it's own Public IP address
* It gets very expensive to have external IP for each of the service (application)

**ExternalName Service**

Maps the Service to the contents of the externalName field (e.g. abc.example.com), by returning a CNAME record with its value. No proxying of any kind is needed. Services of type ExternalName map a Service to a DNS name, not to a typical selector.

A picture containing diagram

Description automatically generated

apiVersion: v1

kind: Service

metadata:

  name: my-service-ext

spec:

  type: ExternalName

  externalName: my2.database.com

When looking up the host **my-service-ext.default.svc.cluster.local**, the cluster DNS Service returns a **CNAME** record with the value **my.database.com**.

Accessing my-service works in the same way as other Services but with the crucial difference that redirection happens at the DNS level rather than via proxying or forwarding. Should you later decide to move your database into your cluster, you can start its Pods, add appropriate selectors or endpoints, and change the Service's type.

Note: Following docker image can be used for debugging networking issues:

kubectl run tmp-shell --rm -i --tty --image **nicolaka/netshoot** -- /bin/bash

**Note:** View the output of **nslookup** my-service-ext.default.svc.cluster.local

**Multi-Port Services**

For some Services, you need to expose more than one port. Kubernetes lets you configure multiple port definitions on a Service object. When using multiple ports for a Service, you must give all of your ports names so that these are unambiguous. For example:

apiVersion: v1

kind: Service

metadata:

  name: my-service-cip

spec:

  type: ClusterIP

  selector:

    app: MyApp

  ports:

    - name: http-port

      protocol: TCP

      port: 8080

      targetPort: 80

    - name: https-port

      protocol: TCP

      port: 44433

      targetPort: 443

**Headless Service:**

* When there is **no need of load balancing** or single-service IP addresses, We create a headless service which is used for creating a service grouping.
* For headless Services that define selectors, the endpoints controller creates Endpoints records in the API, and modifies the DNS configuration to return A records (IP addresses) that point directly to the Pods backing the Service.
* Headless Services are also assigned a DNS record for the name of the form **my-srv.default.svc.cluster.local**. Unlike normal Services, this resolves to the set of IPs of the Pods selected by the Service.

**Headless.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

spec:

  replicas: 10

  selector:

    matchLabels:

      app: nginx-app

  template:

    metadata:

      labels:

        app: nginx-app

    spec:

      containers:

      - name: mynginx-container

        image: nginx

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

**name: my-srv**

spec:

  type: ClusterIP

  selector:

**app: nginx-app**

  clusterIP: None

kubectl apply -f Headless.yaml

kubectl get endpoints

kubectl describe endpoints my-srv

kubectl run tmp-shell --rm -it --image **nicolaka/netshoot** -- /bin/bash

**# nslookup** my-srv.default.svc.cluster.local

# **ping** my-srv.default.svc.cluster.local # resolves to any pod in the service.

**# ping** 10-1-10-90.my-srv.default.svc.cluster.local

**Service Discovery and DNS for Services and Pods**

* Built-in service discovery makes it easier for applications to find and communicate with each other on Kubernetes clusters, even when pods and services are being created, deleted, and shifted between nodes.
* Kubernetes clusters automatically configure an **internal DNS service** to provide a lightweight mechanism for service discovery.
* Kubernetes DNS schedules a **DNS Pod and Service on the cluster**, and configures the **kubelets** to tell individual containers to use the DNS Service's IP to resolve DNS names.
* Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name.

We can see the domain name for the cluster (**cluster.local**)

**kubectl get configmaps coredns -n kube-system -o yaml**

**Services:**

* The full DNS A record of a Kubernetes service will look like **my-service.<namespace>.svc.cluster.local.** This resolves to the **cluster I**P of the Service.
* Assume a Service named **foo** in the Kubernetes namespace **n1**. A Pod running in namespace n1 can look up this service by simply doing a DNS query for **foo**. A Pod running in **namespace n2** can look up this service by doing a DNS query for **n1.foo**.
  + **From pod in n1:** foo **or** foo.n1 **or** foo.n1.svc.cluster.local
  + **From pod in n2:** foo.n1 **or** foo.n1.svc.cluster.local

**Pods:**

* A pod would have a record in this format, reflecting the actual IP address of the pod :

**<pod-ip-address>.<namespace>.pod.cluster.local**

172-17-0-3.default.**pod**.cluster.local

* Any Pods exposed by a Service have the following DNS resolution available:

pod-ip-address.service-name.my-namespace.svc.cluster.local

172-17-0-3.**service-name**.default.**svc**.cluster.local

**Pods and Headless Service**

* The Pod spec has an optional **hostname** field, which can be used to specify the Pod's hostname. When specified, it takes **precedence** over the Pod's name to be the hostname of the Pod. For example, given a Pod with hostname set to "**host1**", the Pod will have its hostname set to "host1".
* The Pod spec also has an optional **subdomain** field which can be used to specify its subdomain.
* If there exists a **headless Service** in the same namespace as the Pod and with the same name as the subdomain, the cluster's DNS Server also returns an A or AAAA record for the Pod's fully qualified hostname.

For example, a Pod with hostname set to "host1", and subdomain set to "sd", and headless service named my-srv, all in namespace "default ", will have the fully qualified domain name (FQDN): **host1.**my-srv**.default.svc.cluster.local**

apiVersion: v1

kind: Service

metadata:

**name: my-srv**

spec:

  selector:

    app: **nginx-app**

  clusterIP: None

---

apiVersion: v1

kind: Pod

metadata:

  name: host1

  labels:

    app: **nginx-app**

spec:

  containers:

  - name: host1-con

    image: nginx:latest

    ports:

    - containerPort: 80

**hostname: host1**

**subdomain:** my-srv

---

apiVersion: v1

kind: Pod

metadata:

  name: host2

  labels:

    app: **nginx-app**

spec:

  containers:

  - name: host2-con

    image: nginx:latest

    ports:

    - containerPort: 80

**hostname: host2**

**subdomain: my-srv**

**Demo:**

**kubectl run tmp-shell --rm -it --image nicolaka/netshoot -- /bin/bash**

**# ping** host1.my-srv.default.svc.cluster.local # resolves to IP address of host1 pod.

**# ping** host2.my-srv.default.svc.cluster.local # resolves to IP address of host2 pod.

**# ping** my-srv.default.svc.cluster.local # resolves to IP address of any pod in the service.

**Note:**

If Pods are deployed through Deployment:

Ping host1.my-srv.default.svc.clouster.local # resolves to **any IP** address in the service

**Blue Green Deployment**

**Use this deployment strategy to test the new version of the application by a set of people in our organization in Production environment before its made available to public on internet.**

Icon

Description automatically generated

**Step1:**

1. Existing Deployment (Blue)

2. Existing Service (Blue)

apiVersion: apps/v1

kind: Deployment

metadata:

  name: deployment-blue

spec:

  replicas: 10

  selector:

    matchLabels:

      app: app-blue

  template:

    metadata:

      name: nginx-pod

      labels:

        app: app-blue

    spec:

      containers:

      - name: mynginx-con

        image: nginx:1.18.0

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: my-service-blue

spec:

  type: NodePort

  selector:

    app: app-blue

  ports:

    - protocol: TCP

      nodePort: 32000

      port: 8080

      targetPort: 80

Open in Browser: http://<minikubeIP>:32000

**Step2:**

3. New Deployment (Green)

Clone the old deployment YAML and change Deployment Name, Label and Image

4. New Service (Green)

Clone the old service and change Name, Selector Label and and Port

apiVersion: apps/v1

kind: Deployment

metadata:

  name: deployment-green

spec:

  replicas: 10

  selector:

    matchLabels:

      app: app-green

  template:

    metadata:

      name: nginx-pod

      labels:

        app: app-green

    spec:

      containers:

      - name: httpd-con

        image: httpd

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: my-service-green

spec:

  type: NodePort

  selector:

    app: app-green

  ports:

    - protocol: TCP

      nodePort: 32002

      port: 8080

      targetPort: 80

Step3: Test of New Deployment using New Service (**http://<MinikubeIP>:32001)**

Step4: Move Traffic from Blue(Existing) to Green(New) Deployment

1. Change **Selector** of existing blue service to New Deployment Label (app: app-green)

apiVersion: v1

kind: Service

metadata:

  name: my-service-blue

spec:

  type: NodePort

  selector:

    app: app-green #Change from app-blue to app-green

  ports:

    - protocol: TCP

      port: 8080

      targetPort: 80

1. Open in Browser: http://<minikubeIP>:32000 and note that the o/p is from New Deployment
2. **Green Deployment image can be replaced with older version (nginx)**
3. **If end users face issues**
   1. **Restore the label of blue to blue deployment**

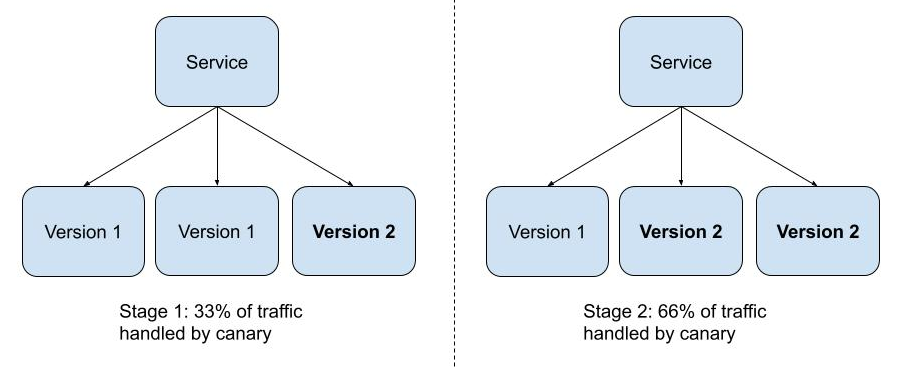
**else**

* 1. **Change the blue Deployment with New Image (httpd) and Blue service label should be restored to blue**
  2. **Delete green deployment and green service as you are convinced that new vesion is good for end users.**

**Canary Deployment**

**Gradually** move load from existing deployment to New Deployment.

1. Create New Deployment with Same Label as Existing Deployment.
2. Gradually increase replicas of New and decrease replicas of Existing.



apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

spec:

  replicas: 10

  selector:

    matchLabels:

      app: nginx-app

  template:

    metadata:

      name: nginx-pod

      labels:

        app: nginx-app

    spec:

      containers:

      - name: mynginx-con

        image: nginx:1.16.0

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: my-service-lb

spec:

  type: NodePort

  selector:

    app: nginx-app

  ports:

    - protocol: TCP

      nodePort: 32000

      port: 8080

      targetPort: 80

---

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment-canary

spec:

  replicas: 2

  selector:

    matchLabels:

      app: nginx-app

  template:

    metadata:

      name: nginx-pod

      labels:

        app: nginx-app

    spec:

      containers:

      - name: mynginx-con

        image: httpd

        ports:

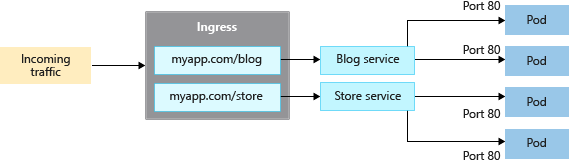
        - containerPort: 80

With time reduce the replicas of Main Deployment and increase the replica of Canary Deployment.

Do this as long as the users are not complaining about the new deployment and old deployment replica changes to "0" and all traffic is shifted to Canary Deployment which now should become Main deployment.

**Ingress Controller**

* Ingress exposes **HTTP and HTTPS** routes from **outside the cluster** to services within the cluster. Traffic routing is controlled by **rules** defined on the Ingress resource.
* Ingress actually acts as a proxy to bring traffic into the cluster, then uses internal service routing to get the traffic where it is going.
* An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting.
* **Ingress controllers** work **at layer 7** (unlike LoadBalancer Service which works at layer 4), and can use more intelligent rules to distribute application traffic.



A screenshot of a computer

Description automatically generated with low confidence

In order for the Ingress resource to work, the cluster must have an ingress controller running.

(List of Ingress Controllers: <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>)

**Ingress with NGINX Ingress Controller** as a reverse proxy and load balancer **(**[**https://kubernetes.github.io/ingress-nginx/deploy/**](https://kubernetes.github.io/ingress-nginx/deploy/)**).**

**Step1:** Install NGINX Ingress controller

kubectl apply -f <https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.0/deploy/static/provider/cloud/deploy.yaml>

OR  
For Minikube: **minikube addons enable ingress**

**Step2:** Create a YAMLs file as below

**Nginx.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: mynginx

spec:

replicas: 1

selector:

matchLabels:

app: mynginx

template:

metadata:

labels:

app: mynginx

spec:

containers:

- image: nginx

name: mynginx

---

apiVersion: v1

kind: Service

metadata:

name: mynginx-cip

spec:

ports:

- port: 8090

protocol: TCP

targetPort: 80

selector:

app: mynginx

**httpd.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

app: myhttpd

name: myhttpd

spec:

replicas: 1

selector:

matchLabels:

app: myhttpd

template:

metadata:

labels:

app: myhttpd

spec:

containers:

- image: **httpd**

name: myhttpd

---

apiVersion: v1

kind: Service

metadata:

name: myhttpd-cip

spec:

ports:

- port: 8090

protocol: TCP

targetPort: 80

selector:

app: myhttpd

**Ingress.yaml**

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: my-ingress

  annotations:

    nginx.ingress.kubernetes.io/rewrite-target: /

spec:

  ingressClassName: nginx

~~defaultBackend:~~

~~service:~~

~~name: mynginx-cip~~

~~port:~~

~~number: 8090~~

  rules:

  - http:

      paths:

      - path: /nginx

        pathType: Prefix

        backend:

          service:

            name: mynginx-cip

            port:

              number: 8090

      - path: /httpd

        pathType: Prefix

        backend:

          service:

            name: myhttpd-cip

            port:

              number: 8090

      - path: /\*

        pathType: Prefix

        backend:

          service:

            name: myhttpd-cip

            port:

              number: 8090

**kubectl apply -f Ingress.yaml**

kubectl get ingress --watch #Wait till shows the IP

curl http://<IngressIP>/nginx

curl http://<IngressIP>/httpd

curl http://<IngressIP>/Random

**Rules based on Domain name:**

rules:

**- host: nginx.mydomain.com**

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: mynginx-cip

port:

number: 8090

**- host: httpd.mydomain.com**

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: myhttpd-cip

port:

number: 8090

Note: Stop IIS or Apache or Tomcat or any other service running on your machine on Port 80.

OR

curl --header 'Host: nginx.mydomain.com' http://<ingress ip>:80

curl --header 'Host: httpd.mydomain.com' http://<ingress ip>:80

OR

Edit as administrator:

**Windows**: c:\windows\system32\drivers\etc\hosts

**Mac/Linux**: /etc/hosts

sudo nano /etc/hosts

Edit file as below and save.

<use ip shown above from minikube ip> nginx.mydomain.com

<use ip shown above from minikube ip> httpd.mydomain.com

**To View the Logs and the POD to which the traffic is forward:**

kubectl get pods **-n ingress-nginx**

kubectl logs **-n ingress-nginx** pod/ingress-nginx-controller-XXXXXXX-xxxx

**Configure Ingress TLS/SSL Certificate**

Diagram

Description automatically generated

1. Create self-signed certificate

openssl req -x509 -newkey rsa:4096 -sha256 -nodes -subj "/CN=demo.mydomain.com" -days 365 -keyout **tls\_self.key** -out **tls\_self.crt**

1. Create Kubernetes Secret from the Key and Cert files

kubectl create **secret** tls ingress-app-tls --key **tls\_self.key** --cert **tls\_self.crt**

1. Add the TLS Block to Ingress Object under spec:

  tls:

    - hosts:

      - demo.mydomain.com

      secretName: ingress-app-tls

1. Browse <https://demo.mydomain.com/nginx>
2. Even if we use the ‚http://‘ prefix in the Browser, the Kubernetes ingress controller will redirect us to the HTTPS counterpart. We can see that easily by using curl

curl <http://demo.mydomain.com>