Links:

<https://archive.org/details/dev-ops-bootcamp_202201/%5BTutsNode.com%5D+-+DevOps+Bootcamp/lesson109.mp4>

<https://www.youtube.com/watch?v=X48VuDVv0do>

<https://gitlab.com/nanuchi/youtube-tutorial-series/-/tree/master/demo-kubernetes-components>

https://gitlab.com/nanuchi/youtube-tutorial-series/-/blob/master/demo-kubernetes-components/mongo-express.yaml

**Definitie**

Kubernetes - open source container orchestration tool (docker/other technologies

> Helps you manage containerized applications in different deployment environments (physical and virtual machine or hybrid environment)

The need of Kubernetes appeared along the trend from monolith to Microservices due to the increased usage of containers - demand for a proper way of managing those hundreds of containers

Orchestration tools like Kubernetes offer

> High availability or no downtime - always accessible by the users

> Scalability or high performance - it loads fasts and users have high response rates

> Disaster recovery - backup and restore (to the latest state)

**Components of the cluster**

**A. Worker node**

Node - Worker machine in K8s Cluster

> Each node has multiple Pods on it

> 3 processes must be installed on every node

> Worker Nodes do the actual work

Processes:

> container runtime (Docker) - app pods have containers running inside

> Kubelet - the process that schedules the pods and the containers - a process of K8s itself - interacts with both - the container and the nod. Kubelet starts the pod with a container inside and assigns resources from the node to the pods.

> Kube proxy - forwarding requests from services to pods. If an app replica it’s making a request do the database, instead of a service forwarding a request to another replica it forwarded the request to the pod in the same nod

1. Pod smallest unit of K8s is an abstraction over container - it creates a running environment

> Usually 1 app per Pod

> Each Pod gets its own IP address - are ephemeral and are gone at pod restart and gets a new Ip address on re-creation

> Pod: abstraction of containers

2. Service - communicate between the Pods - it’s a permanent IP address and acts as a load balancer

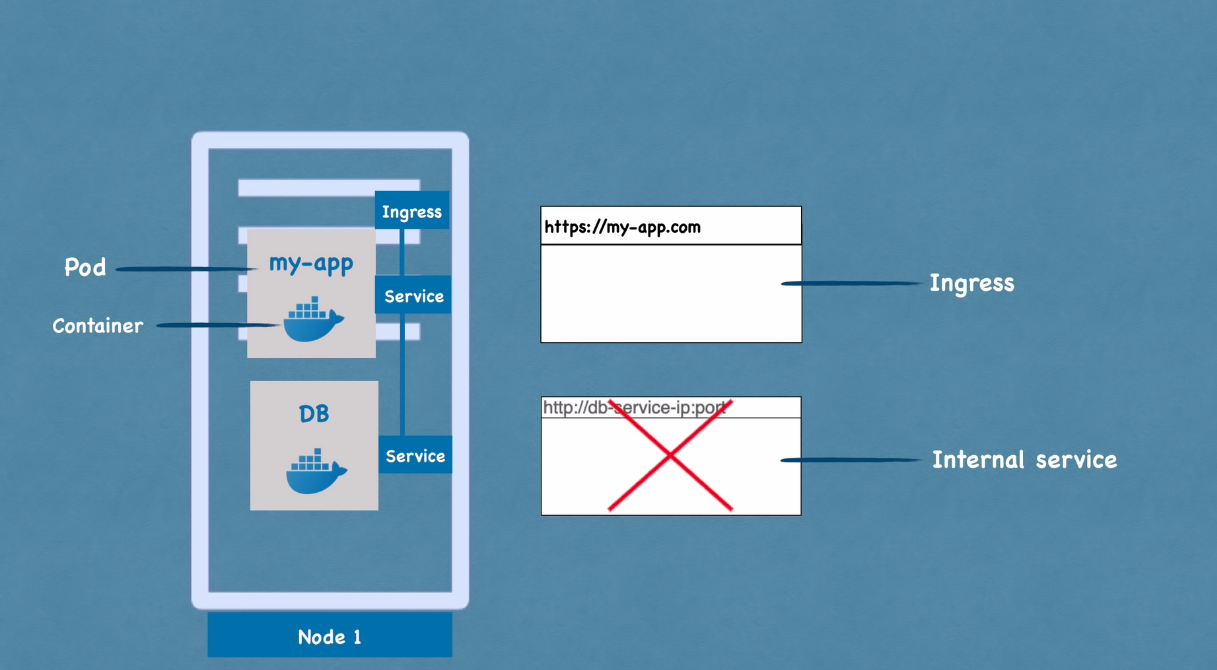
> lifecycle of Pod and Service are not connected

http://nodeiP:port

Http:

External service - open communication from external sources

Internal service - you have to create an internal service



3. Ingress - route traffic into cluster

4. ConfigMap - External configuration: external configuration of your application

DB\_URL = mongo-db-service

DB\_URL = mongo-db

Nu mai trebuie sa modifici serviciu, confi map u tre sa se ocupe

Nu se pun credentials in ConfigMag (user/password)

5. Secret - External configuration - just like config map, but it’s used to store secret data (credentials - password, certificates) that it in base64 encoded - and just like configmap it may be connected to pods so that pods can see those data and read the secret

You can use the data from configmap/secret inside of your app pod using environment variables or as a properties file

6. Volumes - data storage - Data persistence

> If a pod/container gets restarted the data would be gone

> By using Volumes you avoid this - attach a physical storage on a local machine(on the same server node)/remote (outside of the Kubernetes Cluster - on prem/Cloud Storage - not part of the Kubernetes Cluster)

7. Deployment - blueprint for app pods (you can specify how many replica of pods)

> abstraction of pods which pods are abstraction on containers

> Deployment for sateLESS apps

8. StafefulSet for stateFUL apps or Databases - Database can’t be replicated via Deployment because it has state

> Stateful stat it’s for stateful apps like databases (elastik search, mongo DB, mySQL)

Databases are often hosted outside of K8s Cluster

Deployments and StatefulSet - Pod Blueprints with Replicating mechanisms

**B. Master nodes processes (control plane):**

> API server - the point of interaction between us and the cluster (cluster gateway)

Requests -> API Server -> validates requests -> other processes

Enables interaction with cluster (UI, K8s API, CLI (kubectl - the most powerful)

> Scheduler - decides on which node new pod should be scheduled. The process that actually does the scheduling is the Kubelet that gets the req from the scheduler and executes that request on that Node

Schedule new pod request -> API Server -> Scheduler (decides where to put the Pod, on which Node based on the resources needed

> Controller manager - detects cluster state changes (when pods die) and tries to recover the cluster state as soon as possible

Controller manager request -> to Scheduler to reschedule those dead pods and the scheduler decides based on the resource calculation which Working Nodes should restart those pods again and -> makes request to the Kubelet to actually restart the pods

> etcd - key value store of a cluster state - every change in the cluster get stored in the key value store (etcd)

How does the scheduler hat resources are available? How does the controller manager know the cluster state changed? How does the API server when we make a request that the cluster is healthy? All of this information is storage in the etcd

Application data is NOT stored in the etcd

In practice K8s Cluster is made of multiple masters - API server is load balanced and etcd form a distributed storage across all master nodes

**Example production Cluster Set-up**

2 Master Nodes - more important but they need less resources

3 Worker Nodes - more resources (or multiple worker nodes)

Minikube it’s a one node cluster - has docker pre-installed - master and node run on the same node that runs through Virtual Box (or another hypervisor) and has testing purposes.

Kubectl cli - for configuring the cluster

Minikube - start/delete cluster

**Kubectl commands:**

Kubectl - command line tool for K8s Cluster - create pods/services, destroy them etc

You create deployments and not pods (abstraction over pods)

Replicaset it’s another abstract layer between the pod and the deployment

It’s managing the replicas of a pod

Layers of abstraction

Deployment manages a ReplicaSet that manages all the replicas of a pod and Pod is an abstraction of Container

kubectl exec -it mongo-depl-5ccf565747-cgzmv -- bin/bash

Connect as a root user to that pod (interactive terminal and pod name)

Kubectl apply -f config.yaml (-f = file)

**Config files:**

API version - every type of service has different type of apiversion

kind - deployment/service etc

1. Metadata (name)
2. Specification (every king of configuration that you would like to apply to that component)

Attributes of spec are specific to the kind (deployment/serv etc)

1. Status (automatically generated and added by Kubernetes) -> desired / actual state - self healing

Yaml validator

Store the config file with your code/own git repository

Template - a config file inside the config file - applies to the Pod - blueprint for a Pod (image, port, name)

Labels (metadata part) and selectors (specification part)

Connecting deployments to Pods - any key value pair for component

Labels:

App: nginx

Pods get the label through the template blueprint

This label is matched by the selector:

Selector:

MatchLabels:

App: nginx

The label is matched by the selector

It is the same connecting services to deployments and the deployment pods (through selector of the label)

Service have ports - port and targetport - the target port of the service must be the same of the containerPort in the deployment (template area)

kubectl describe service [SERVICE name] - da informatii utile despre servicii (selector app = … ; TargetPort; Endpoints -> avem ip urile si porturile pt podurile catre care va fi forwardat requestu)

kubectl get pod -o wide (ofera mai multe infor in legatura cu pod urile - IP uri, readiness gate) - asa putem verifica daca ip urile podurilor running sunt aceleasi ca in serviciu, sau care sunt endpoints definite pt serviciu de mai sus

Aflam status generat de k8s - obtinem rezultatul updatat al deploymentului care se afla in ETCD

kubectl get deployment nginx-deployment -o yaml

sau

kubectl get deployment nginx-deployment -o yaml > nginx-deployment-result.yaml

Si putem vedea statusul deploymentului (creation time stamp

Putem sterge si resursele si pe baza de config files yaml

kubectl delete -f nginx-deployment.yaml

kubectl delete -f nginx-service.yaml

**Proiect**

Facem un deployment de 2 aplicatii - mongo-sxpress si mongoDB

Web app - > request to mongoDB

Arhitectura aplicatiei:  
> 2 Deployments / Pod

> 2 Service

> 1 ConfigMap

> 1 Secret

> Cream un pod(Deployment) de MongoDB si ca sa comunicam cu MongoDB cream un serviciu intern (no external requests allowed to the pod - only components inside the same cluster can talk to it)

> Cream un deployment de Mongo Express

> ConfigMap - o sa avem nevoie de database URL al MongoDB ca mongo express sa se poata conecta la el - se face pe baza de deployment.yaml prin environment variables

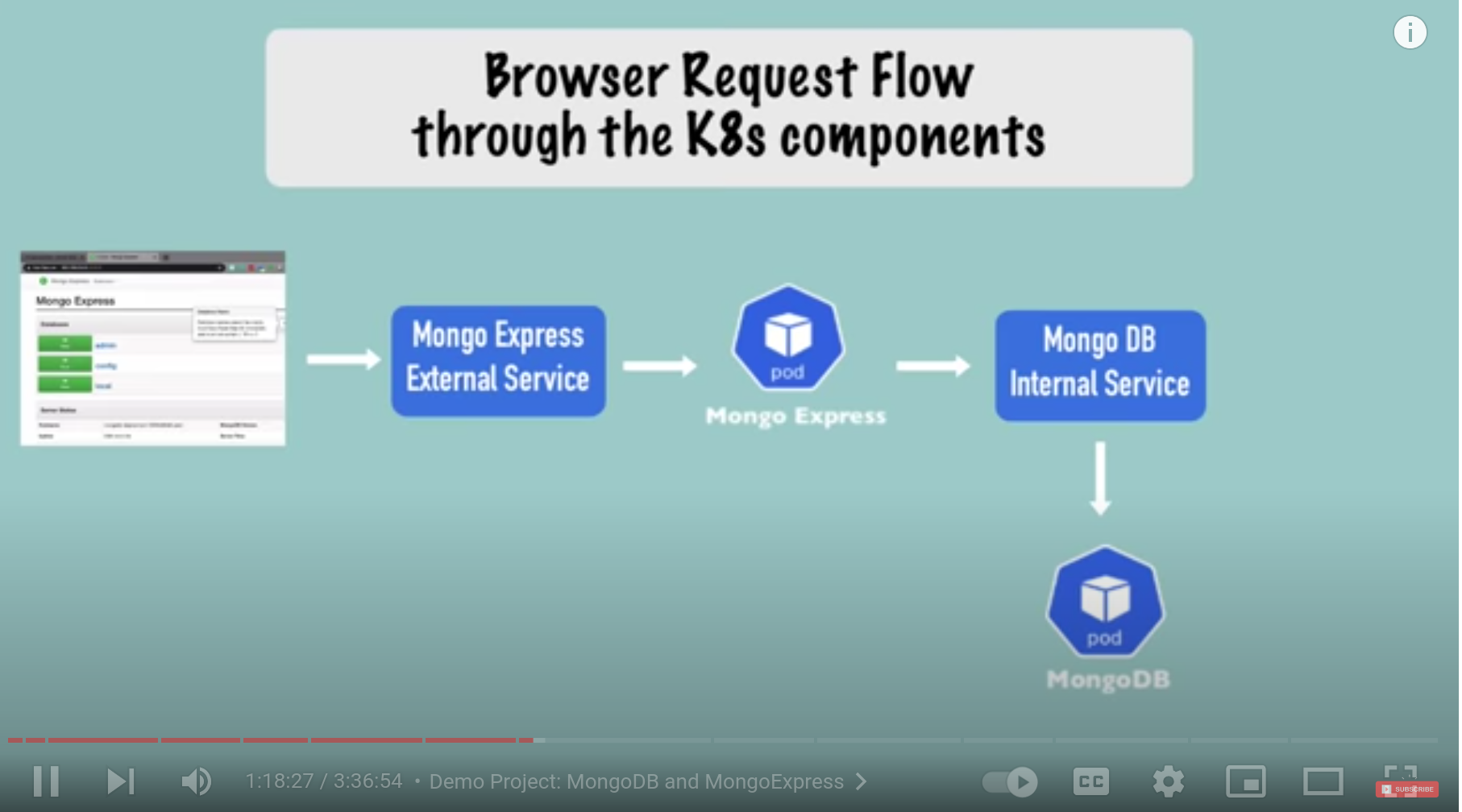
URL:

- IP address of node

- Port of external Service

> Secret - credentials (username and pwd of DB) ca sa se poata conecta - se face pe baza de deployment.yaml prin environment variables

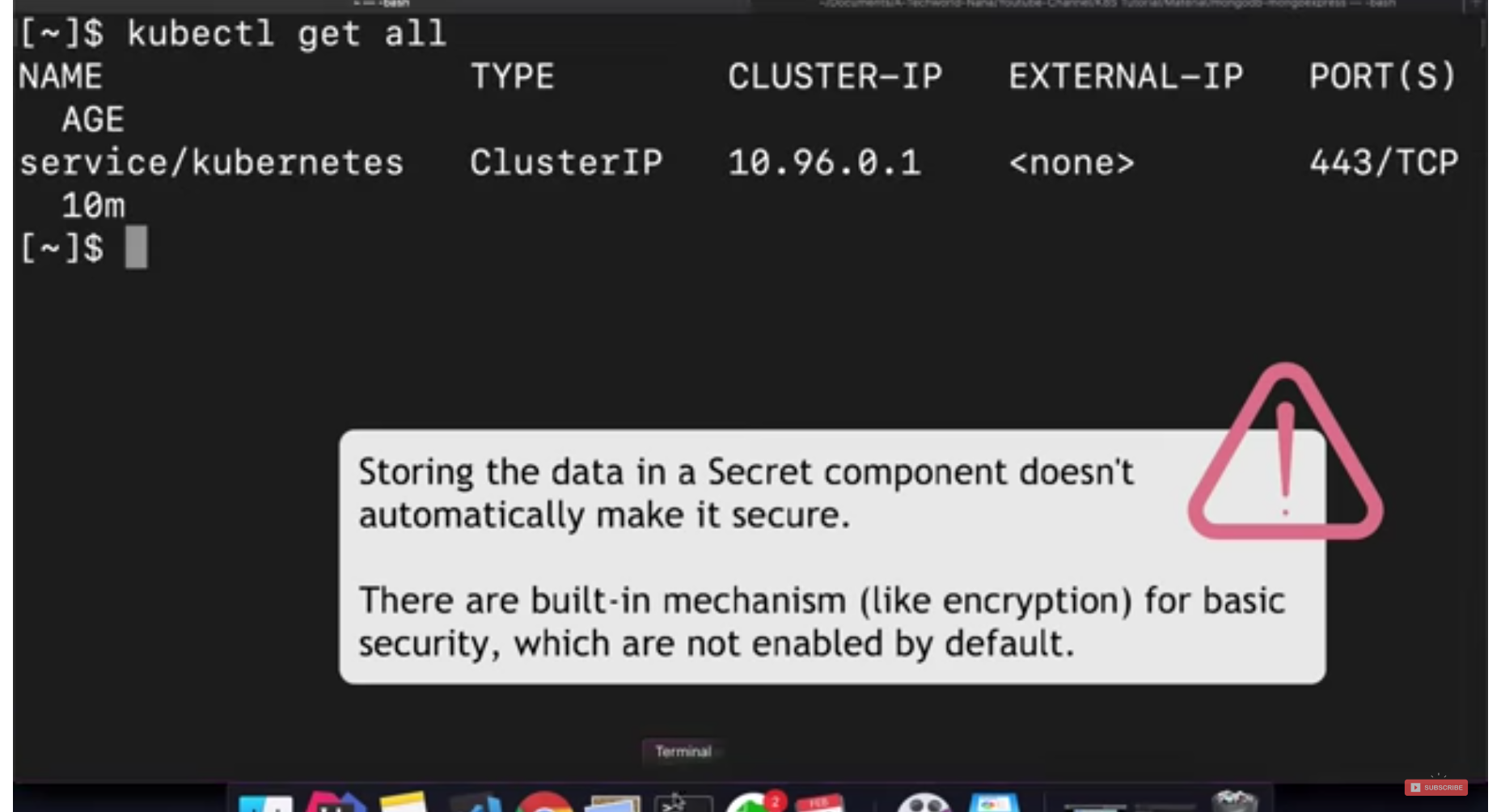
> External Service - pentru a putea accesa mongo express prin browser



1. Vine request din browser

2. Requestu merge catre external service mongo express care forwardeaza traficu catre Mongo Express deployment/pod

3. Pod ul de Mongo Express se va conecta cu Internal Service Mongo DB (DB URL ConfigMap) si il va forwarda catre MongoDB pod unde se va autentifica pe baza de credentials (Secret)



mihailstan@minikube:~$ echo -n 'username' | base64

mihailstan@minikube:~$ echo -n 'password' | base64

Valorile se introduc in secret config ca valori

Secret se face inainte de deployment daca facem referinta in deployment la secret. La fel si la ConfigMap

Cand faci un serviciu de k8s si nu ii dai type, by default il face Internal Service/Cluster IP si va aloca serviciului un internal ip address, in timp ce LoadBalancer type ii da si external

https://gitlab.com/nanuchi/youtube-tutorial-series/-/tree/master/demo-kubernetes-components

In minikube functioneaza diferit si iti da pendind la external ip

mihailstan@minikube:~$ kubectl get service

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 32d

mongo-express-service LoadBalancer 10.99.45.128 <pending> 8081:30000/TCP 3m16s

mongodb-service ClusterIP 10.96.224.12 <none> 27017/TCP 24m

Pt internal ip address (cluster IP) ai nevoie de un singur port

Pt LoadBalancer type ai 3 porturi, pentru ca trebuie al 3lea pt external ip address

Minikube service [service name] - ca sa asignam un ip extern

**K8S Namespaces**

A Namespace - virtual cluster inside a cluster - poti organiza resurse in namespaces, astfel incat sa ai mai multe namespaces intr un cluster

By default, Kubernetes has 4 namespaces already created

Kubectl get namspace

> default - resources you create are located here by default

> kube-node-lease - scopul este ca tine informatie despre heartbeats of nodes, each node has associated lease object in namespace that contains the information of that node availability

> kube-public - publicly accessible data - a configmap which contains cluster information

Daca dam

kubectl cluster-info

ni se da un output

> kube-system - is not meant for our use - do NOT create/modify in kube-system - componentele deployed in asta sunt system processes din master sau kubectl process

> kubernetes-dashboard - only with minikube

Kubectl create namespace [my-namespace]

Kubectl get namespace

You can create a namespace with a configuration file

1. The first usecase of using namespace is that you are having a better overview of complex architecture:  
> Database namespace - deploy database and required resources

> Monitoring namespace - deploy Prometheus

> Elastic Stack namespace - Kibana etc

> Nginx-Ingress namespace

2. Conflicts: two teams, same application - se poate override app ul

3. Resource sharing: staging si development environments in acelasi cluster care acceseaza doua namespace uri (nginx-ingress si elastic de exemplu)

4. Resource Sharing: Blue/Green Deployment pt aplicatii - in acelasi cluster am doua versiuni ale productiei - cea activa care e acum in productie si noul release care acceseaza doua namespace uri (nginx-ingress si elastic de exemplu - shared resources)

5. Access and Resource Limits on Namespaces

> secure isolated environments

> Limit: CPU, RAM, Storage per namespace (resource quota)

La secrete si ConfigMap nu pot fi shareuite intre namespace uri, fiecare namespace trebuie sa aiba ConfigMap si secretul lui care pot accesa servicii din alte namespace uri.

Access Service in another Namespace: in db\_url: [name].database - .database este stamp ul namespace ului

Exista componente in K8s care sunt globale, nu pot fi izolate intr un namespace: volume si noduri - sunt accesibile in tot clusterul

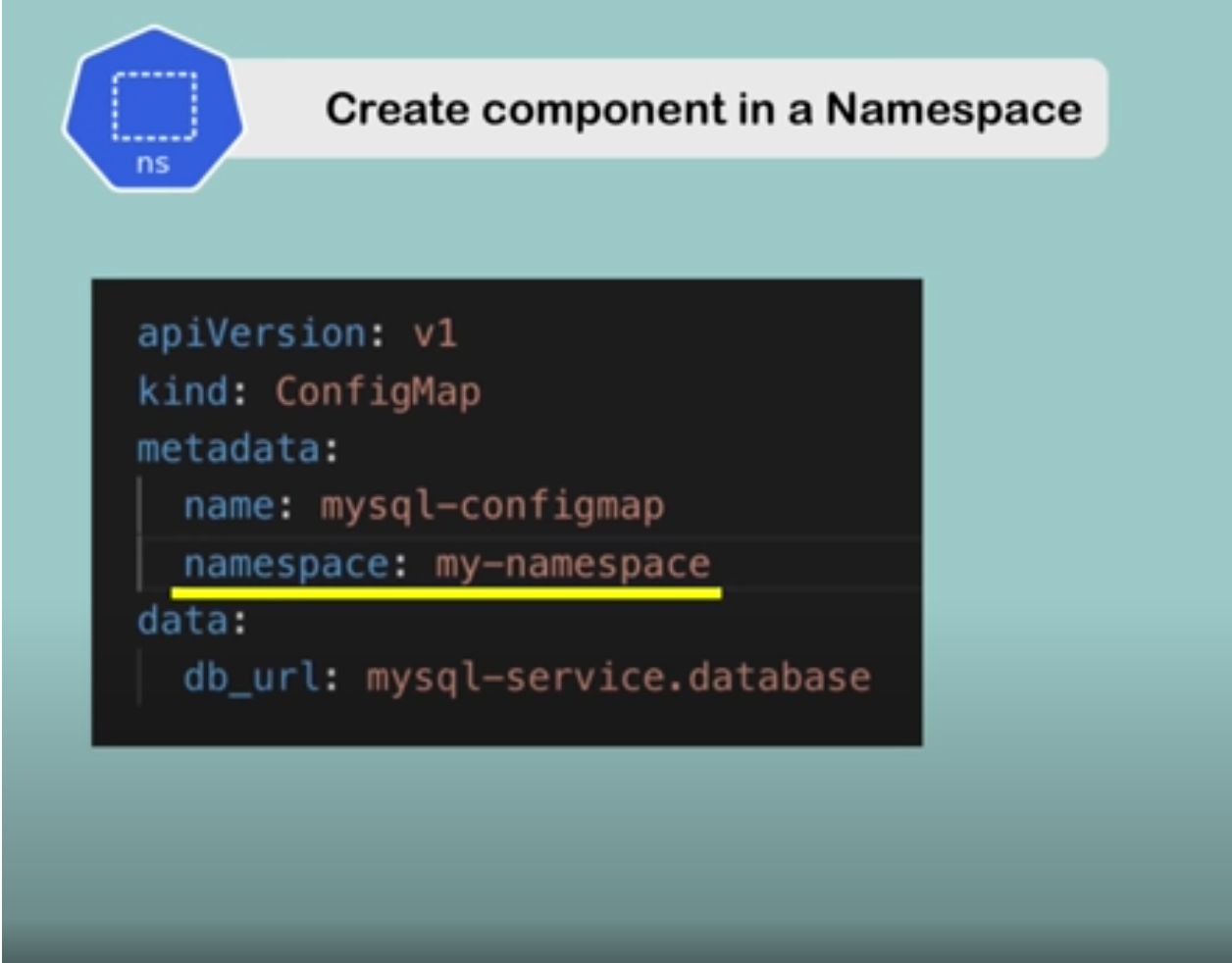
Kubectl api-resources –namespaced=false

Kubectl api-resources –namespaced=true

Putem crea resurse in namespace:

Kubectl apply -f [cfg.yaml] –namespace=[namespace name]

Sau prin configfile



Kubectl get configmap -n [namespace] - comanda asta trebuie data cu -n si numele namespace ului pt ca kubectl get automat cauta in default namespace

Kubens - change active namespace (e un tool care trebuie instalat)

Kubens - list si cu verde namespace u activ

Kubens [namespace name] - creeaza un nou namespace si il face default

**INGRESS**

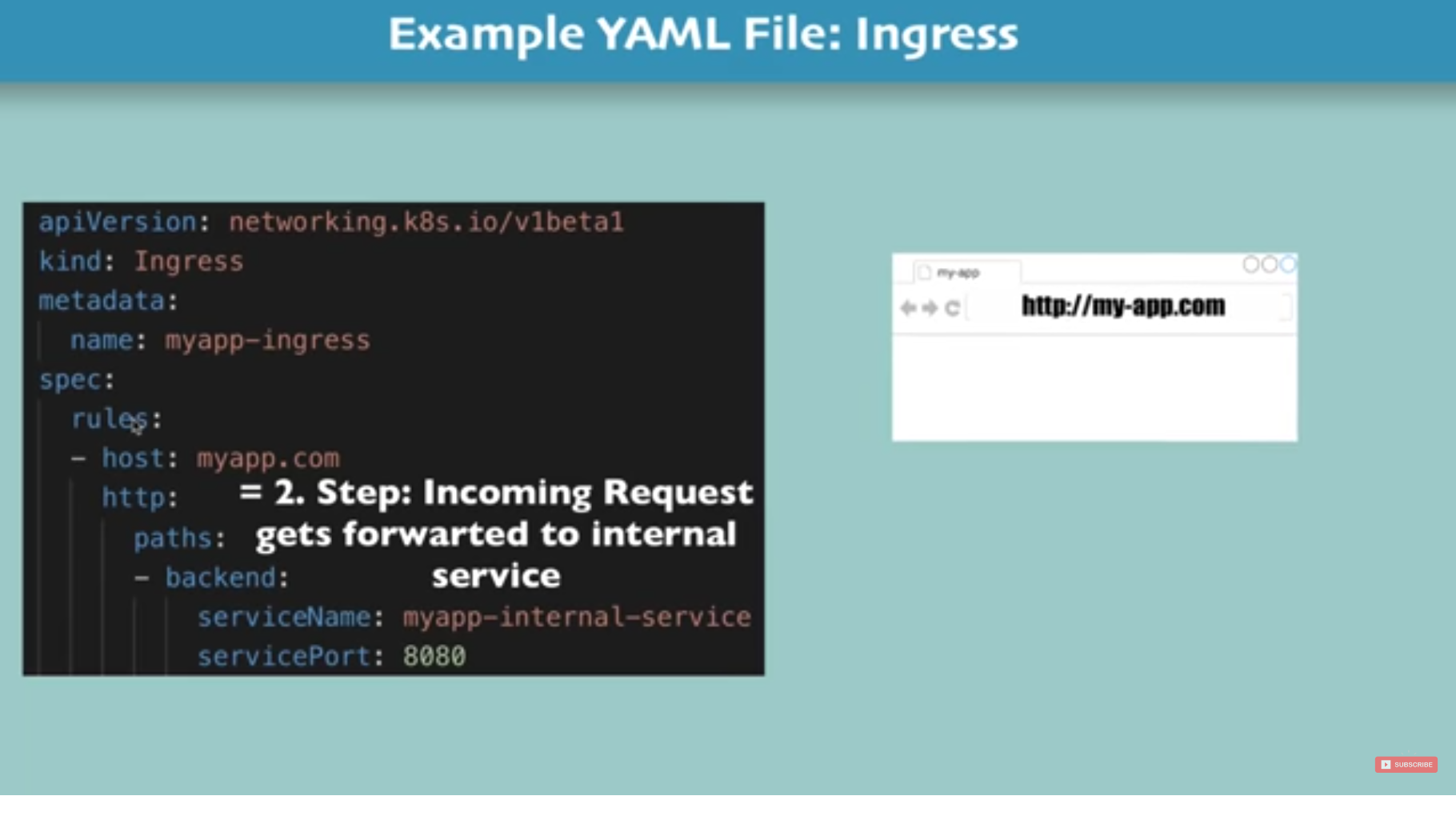
Request browser -> ingress -> internal service -> deployment

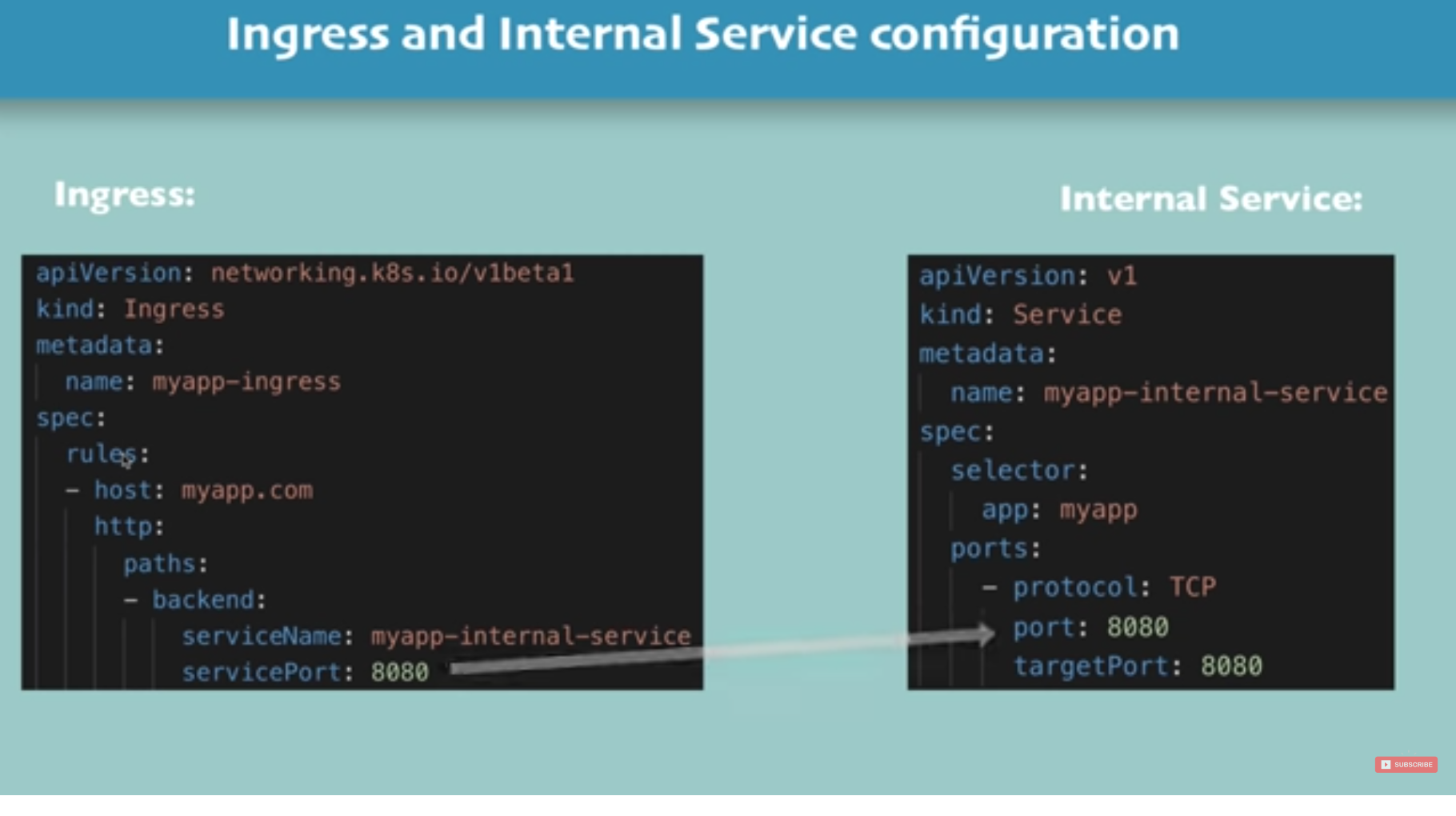
1. Request from browser to ingress

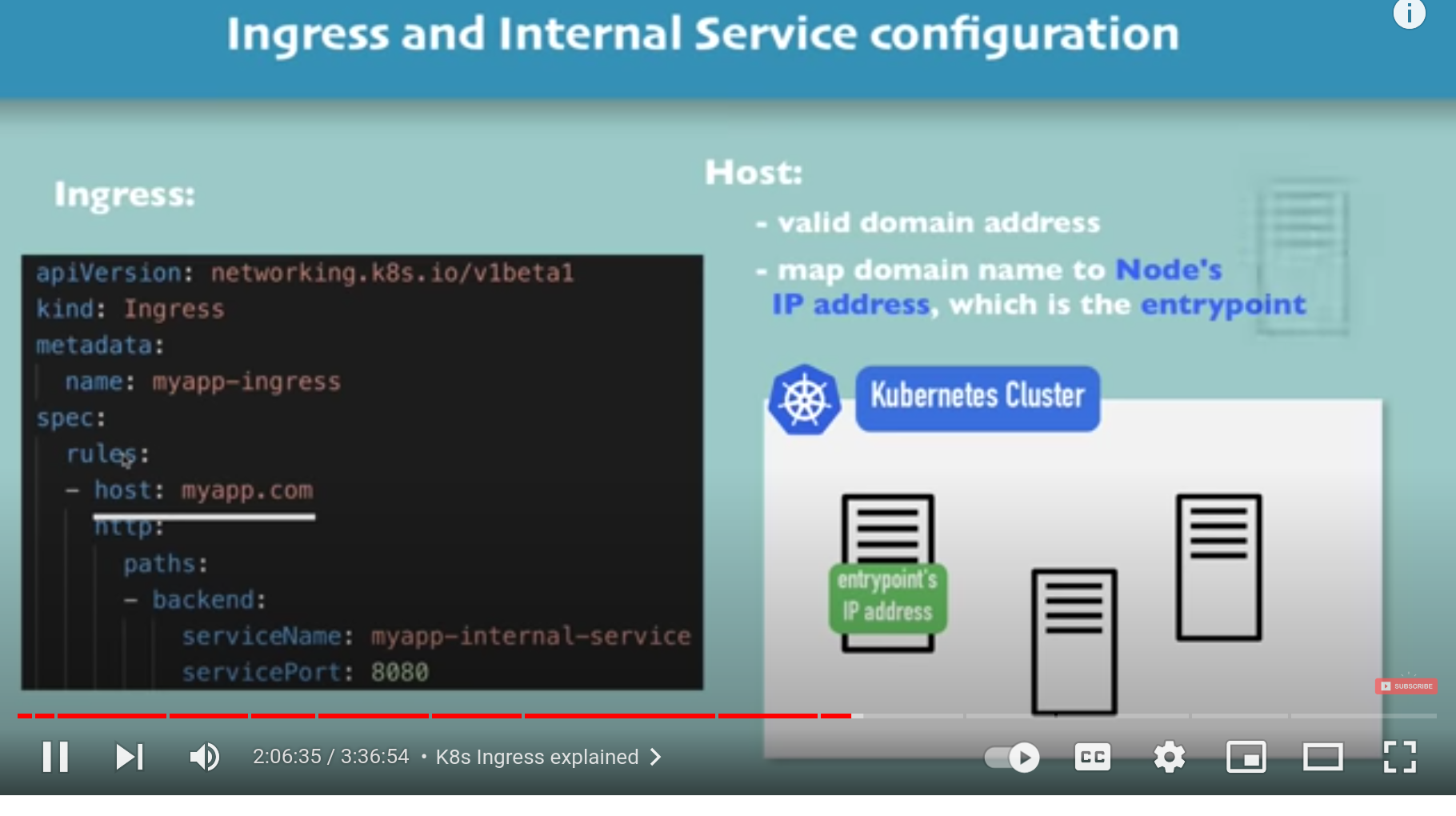
2. Incoming Request gets forwarded to internal servicce

Ingress needs routing rules in order to forward request to the internal service

paths=url path







For Ingress you need an implementation which is Ingress Controller - un alt set de poduri - minutul 2:05

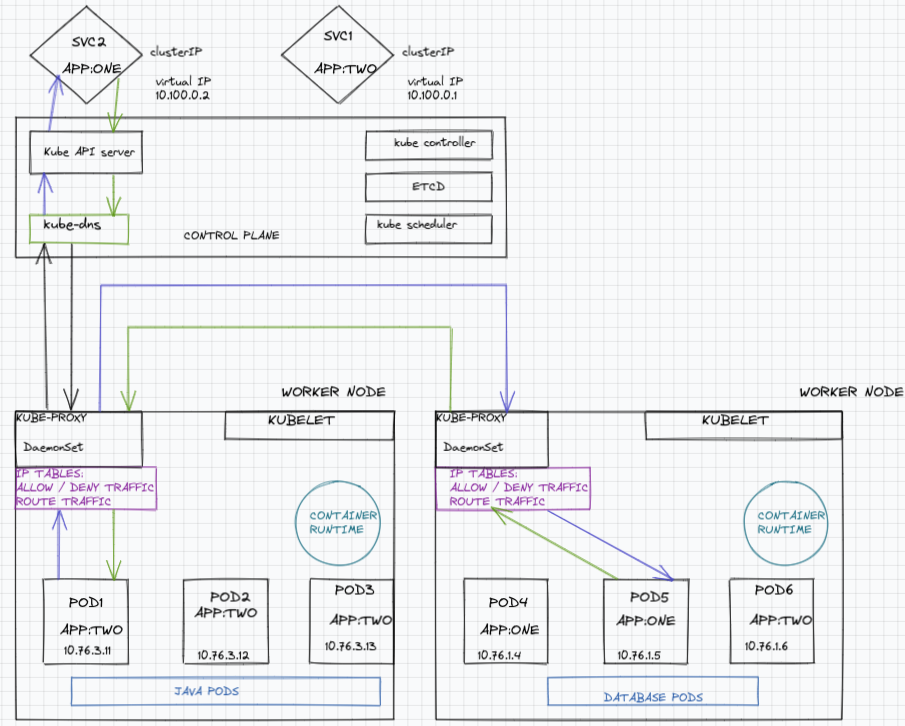
**The life of a pod request in Kubernetes**

One of the major strong points in Kubernetes is the pod networking. When we deploy a pod in a cluster, kube-scheduler is responsible for assigning that pod to a node. Multiple things are taken into consideration for this process, but we will stick to a simple scenario for a better understanding.

In a cluster with two worker nodes, two pods will most likely be deployed in different nodes, as the kube-scheduler is trying to allocate the resources in a balanced manner.

This networking architecture allows a pod from a node to make requests to pods from different nodes without giving you a hard time. As these requests are occurring inside the cluster, they do not have to reach the public internet and the pod will know the source of the request.

Let’s take a simple example with a Java pod that makes a request to the MySQL database.



The service SVC1 is configured to see only the pods labeled as “APP:TWO”, while the service SVC2 only knows about the pods labeled as “APP:ONE” and they will store the IPs of the pods as multiple endpoints. These services are of type ClusterIP, meaning that they only route requests within the cluster and they do not belong to a specific node.

If the POD1 will need some data from a database pod, it will make a GET request to the service SVC2. But how does the pod know how to reach the database? The kube-proxy will query the DNS server, and if this does not have the service’s endpoints cached, it will ask the kube-apiserver. After the service name is resolved, and the kube-proxy has the ip address and the port of the necessary pod based on a round robin algorithm, it will route the request to the kube-proxy of the node that has the desired pod. The kube-proxy will get the information from the database pod and then will send the response back to the kube-proxy that started the communication. This one will finally deliver the requested information to POD1

As you see, this is a pretty long journey, but since the pods are dying all the time, it would be unfeasible to make a request directly to the pod’s IP.

However, since I am not so enthusiastic about theory, I wanted to see it with my eyes so I thought a PCAP file analyzed with Wireshark would be the best approach.

I started by creating 2 pods with an Nginx image and named them tcp-pod and ping-pod. Here are the yaml files:

apiVersion: v1

kind: Pod

metadata:

name: ping-pod

labels:

app: ping

spec:

containers:

- name: ping-container

image: nginx

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: ping-service

spec:

selector:

app: ping

ports:

- name: name-of-service-port

protocol: TCP

port: 82

targetPort: 80

apiVersion: v1

kind: Pod

metadata:

name: tcp-pod

labels:

app: tcp

spec:

containers:

- name: tcp-container

image: nginx

ports:

- containerPort: 80

—

apiVersion: v1

kind: Service

metadata:

name: tcp-service

spec:

selector:

app: tcp

ports:

- name: name-of-service-port

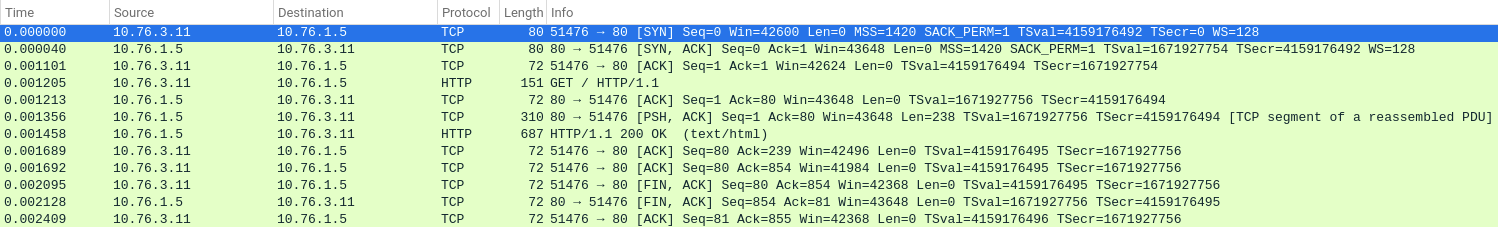
protocol: TCP

port: 81

targetPort: 80

After I exec-ed into the ping pod with “kubectl exec -it ping-pod -- bash”, I installed ping with “apt-get update -y” and “apt-get install -y iputils-ping”. Then I exec-ed into the tcp pod with “kubectl exec -it tcp-pod -- bash” and installed tcpdump with “apt-get update -y” and “apt-get install -y tcpdump”. To start listening and writing the output with tcpdump I used this command “ tcpdump -i any -w ping-request.pcap “. I opened another terminal and exec-ed again into the ping-pod and started to make some requests to the tcp-service that would route the packets to the tcp-pod with a few “curl tcp-service:81” requests. However, you might observe that pinging a service name is pointless, as the IP of the ClusterIP services is a virtual IP and it’s living inside etcd, the cluster database.

Coming back, I closed the tcpdump capture and I analyzed it with Wireshark. This was the output.



We can see that the tcp-pod capture some packets with source: ping-pod

<https://googlecloudcheatsheet.withgoogle.com/architecture?link=a38e5e10-b687-11ed-b21e-198010a38708>

<https://mayankshah.dev/blog/demystifying-kube-proxy/>

<https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/>

