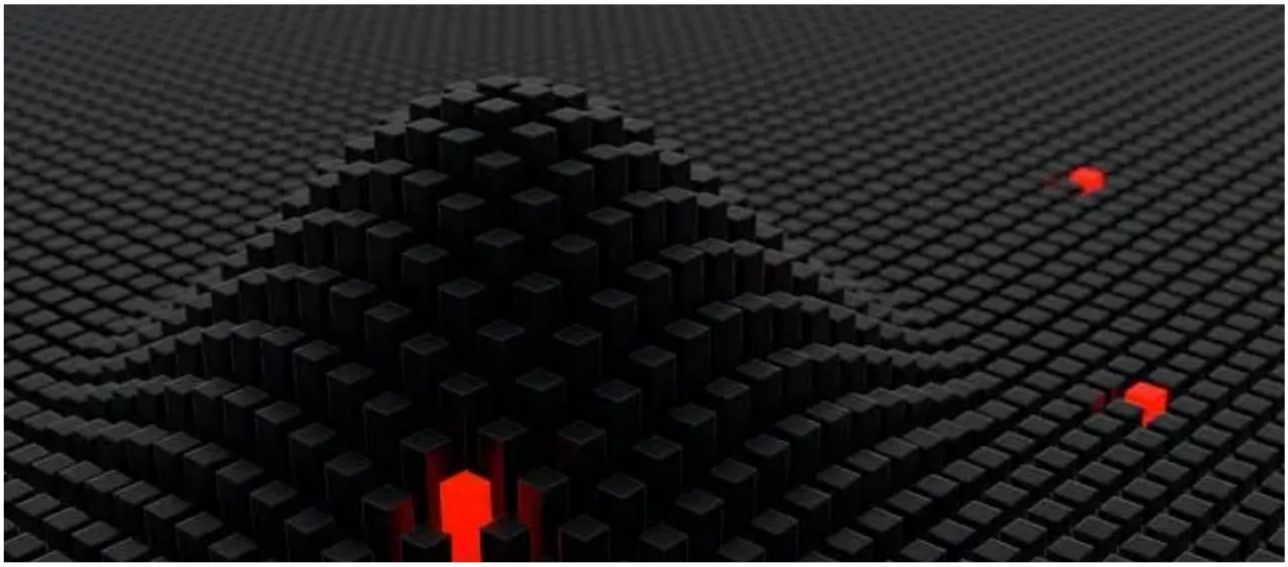
KUBERNETES

NGINX as kubernetes ingress controller



With this step by step guide we are going through the creation of small python Pods up to deploy our minimalistic cluster environment using NGINX as kubernetes ingress controller. The steps below will illustrate steps by steps how to deploy your local or remote cluster, create pods from the docker images and deploy!

What is NGINX

From the NGINX website (https://www.nginx.com/resources/glossary/nginx/)

NGINX *is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. It started out as a web server designed for maximum performance and stability. In addition to its HTTP server capabilities, NGINX can also function as a proxy server for email (IMAP, POP3, and SMTP) and a reverse proxy and load balancer for HTTP, TCP, and UDP servers.*

If you want to know more about NGINX

What is an ingress controller

An ingress controller is usually a load balancer for routing external traffic to your kubernetes cluster. It abstracts the complexity of the network traffic to your kubernetes cluster and redirect the requests to the proper services. In other

ord is a bridge between the requests coming to your applications and the pecific server to which that requests belongs to (we will see more in below sections).

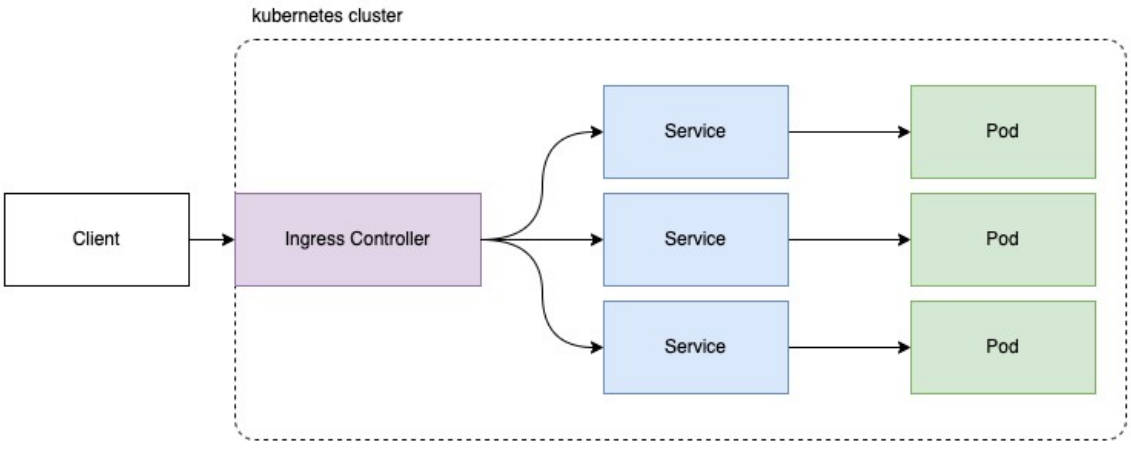
Before continuing...from the immense kubernetes documentation we will need to understand few concepts

Service: *Expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.*

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.*

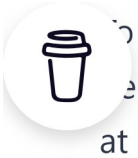
Ingress Controllers: *In order for an Ingress to work in your cluster, there must be an ingress controller running. You need to select at least one ingress controller and make sure it is set up in your cluster. This page lists common ingress controllers that you can deploy.*

This is summarized in the image below



Ingress controller and services relation

Deploy a local kubernetes cluster with some pods



To be able to show how an ingress controller is configured and is working, we need some prerequisites to be met. We need a working kubernetes cluster with at least 2 pods running and a service in front of each one of those. If in doubt at this stage you can use one of the below articles to help you out with the initial configuration

- Install k3s
- Disable traefik (https://keepforyourself.com/kubernetes/disable-traefik-from-k3s/)
- or alternatively you can run through Docker desktop or minikube (https://keepforyourself.com/docker/run-a-kubernetes-cluster-on-your-pc/)
- Deploy some pods https://keepforyourself.com/docker/kubernetes-pod-using-a-local-image/
- Or you can deploy a python pod with FastAPI by creating a local image and deploy to kubernetes as explained below step by step

Create your Images

First step is to create a docker image of some sort of application, this image will be used later as reference for creating the pods. For achieving that, let's go step by step through this image creation and start to create a folder "pod-example" and put in it a Dockerfile with this content

Dockerfile

```
FROM python:3.9
WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt
COPY ./app /code/app
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
EXPOSE 8080
```

Create a requirements.txt

requirements.txt

```
fastapi
```

```
2   pydantic
3   uvicorn
```

Create a main.py in a app folder

pod-example/app/main.py

```
1   from fastapi import FastAPI
2
3   app = FastAPI(
4       title="Example-pod",
5       description="keep-4-yourself-example-pod",
6   )
7
8   @app.get("/pod1")
9   def say_hello():
10      return {"hello": "world 1"}
```

then build your local image

Build your local image

build the image

```
docker build -t local-fastapi-example-pod-1 .
```

You can repeat the process by changing main.py to be

pod-example/app/main.py

```
from fastapi import FastAPI

app = FastAPI(
    title="Example-pod",
    description="keep-4-yourself-example-pod",
)

@app.get("/pod2")
def say_hello():
    return {"hello": "world 2"}
```

o that we can have 2 different images (for testing purpose) and build using the ommand

build second docker image

```
docker build -t local-fastapi-example-pod-2 .
```

[Optional] Build your images with Minikube

If you are running your cluster using minikube you should build the docker images by using the minikube command so it will be immediately available in the minikube images repository

```
minikube image build -t pod1 .
```

[Optional] Import the images

Or alternatively if your kubernetes cluster is not running within docker (like the docker desktop feature) you must export the image and import. In my case I'm running a VM with k3s on it and what I had to do to make the image available in the k3s registry was to manually import the docker image once exported with these commands (as explained here https://keepforyourself.com/docker/import-a-docker-image-in-kubernetes/)

main.py message change

```
# export the images
docker save local-fastapi-example-pod-1 > pod1.tar
docker save local-fastapi-example-pod-2 > pod2.tar

# once the images are exported
sudo k3s ctr image import pod1.tar
sudo k3s ctr image import pod2.tar
```

Create your Namespace

Now let's create our namespace, create a file keep_for_yourself.yaml with the content below

kubernetes namespace creation

```
kind: Namespace
apiVersion: v1
metadata:
  name: k4y
  labels:
    name: k4y
```

Then apply by running

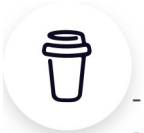
```
$> kubectl apply -f keep_for_yourself.yaml
namespace/k4y created

# you can doublecheck that was created by typing

$> kubectl get namespace
NAME              STATUS   AGE
default           Active   2d20h
k4y               Active   59s
kube-node-lease   Active   2d20h
kube-public       Active   2d20h
kube-system       Active   2d20h
```

Deploy the Pods

Up to here we have created our docker images and a namespace in kubernetes, now we are going to create the two pods by using as image the images we have created earlier here. For doing that extend the keep_for_your_self.yaml file by adding two other sections for the pods. The end result should be something similar to the file below

kubernetes yaml file for 2 pods and 1 namespace

```
kind: Namespace
apiVersion: v1
metadata:
  name: k4y
```



```
  labels:
    name: k4y
---
apiVersion: v1
kind: Pod
metadata:
 name: pod1
 namespace: k4y
 labels:
   app: pod1
spec:
 containers:
 - name: pod1
   image: local-fastapi-example-pod-1
   imagePullPolicy: Never
 restartPolicy: Never

---
apiVersion: v1
kind: Pod
metadata:
 name: pod2
 namespace: k4y
 labels:
   app: pod2
spec:
 containers:
 - name: pod2
   image: local-fastapi-example-pod-2
   imagePullPolicy: Never
 restartPolicy: Never
```

Give it an apply by running the apply command

kubectl apply

```
$> kubectl apply -f kay.yaml
namespace/k4y unchanged
pod/pod1 created
pod/pod2 created
```

We can double check by running this command

```
$> kubectl get pods -n k4y
NAMESPACE     NAME                READY    STATUS    RESTARTS          AGE
k4y           pod1                1/1      Running   0                 7s
```

```
k4y           pod2                1/1     Running   0                 7s
```

Deploy the services

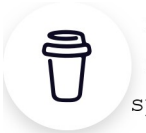
This is great so far! However it won't help us too much because we might want to have access to these 2 pods and for doing that we must put the services in front of them (**remember, the services help to** *Expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.*). For achieving that we need to add this piece to the yaml file we were working on

service definition

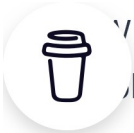
```
kind: Service
apiVersion: v1
metadata:
  name: <the name we want to give to our service>
  namespace: <the namespace in which this service must be>
spec:
  ports:
  - port: <port we want to expose>
    protocol: TCP
    targetPort: <port of the pod we want to hit
  selector:
    app: <the pod name>
---
```

In our case then we will have 2 services, one for each pod, our yaml will look like the one below

```
kind: Namespace
apiVersion: v1
metadata:
  name: k4y
  labels:
    name: k4y

---
apiVersion: v1
kind: Pod
metadata:
```



```
 name: pod1
 namespace: k4y
spec:
 containers:
 - name: pod1
   image: local-fastapi-example-pod-1
   imagePullPolicy: Never
 restartPolicy: Never

---
apiVersion: v1
kind: Pod
metadata:
 name: pod2
 namespace: k4y
spec:
 containers:
 - name: pod1
   image: local-fastapi-example-pod-2
   imagePullPolicy: Never
 restartPolicy: Never
---
kind: Service
apiVersion: v1
metadata:
  name: pod1-service
  namespace: k4y
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: pod1
---
kind: Service
apiVersion: v1
metadata:
  name: pod2-service
  namespace: k4y
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: pod2
---
```



What is still missing is our ingress, so let's go and install our NGINX ingress by following the next step

Install NGINX

For installing NGINX I would strongly recommend to give it a read to this article https://kubernetes.github.io/ingress-nginx/deploy/ it explains case by case what is needed. However the most used command should be

install NGINX ingress

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controll
```

If you are having issues around ip that is **pending** you can patch your service (for learning purpose only) by using this command

```
kubectl patch svc SERVICE_NAME -p '{"spec": {"type": "LoadBalancer", "externalIPs":[
# in my case the command was
kubectl patch svc ingress-nginx-controller -p '{"spec": {"type": "LoadBalancer", "ex
```

Ingress YAML definition

At this stage we have now all the building blocks, what is missing is the ingress definition. The ingress definition will have some key things that is important to mention:

- each path has a match, a type and to which backend service should point to
- the host is any host string we want to use, as long is being added as entry in the /etc/hosts file (in my hosts file I have 192.168.1.56 local-dev-env)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: local-ingress
  namespace: k4y
spec:
```

```
ingressClassName: nginx
rules:
  - host: local-dev-env
    http:
      paths:
        - path: /pod1
          pathType: Prefix
          backend:
            service:
              name:  pod1-service
              port:
                number: 80
        - path: /pod2
          pathType: Prefix
          backend:
            service:
              name:  pod2-service
              port:
                number: 80
```

Now we have all the building blocks needed for our minimal deployment:

Namespace definition

Pods definition (pod1 and pod2)

Services definition (pod1-service and pod2-service)

Ingress definition

and our final yaml file will be like the one below

Final YAML – NGINX ingress controller with 2 pods

YAML deployment example

```
kind: Namespace
apiVersion: v1
metadata:
```

```
  name: k4y
  labels:
    name: k4y
---
apiVersion: v1
kind: Pod
metadata:
 name: pod1
 namespace: k4y
 labels:
   app: pod1
spec:
 containers:
 - name: pod1
   image: local-fastapi-example-pod-1
   imagePullPolicy: Never
 restartPolicy: Never
---
apiVersion: v1
kind: Pod
metadata:
 name: pod2
 namespace: k4y
 labels:
   app: pod2
spec:
 containers:
 - name: pod2
   image: local-fastapi-example-pod-2
   imagePullPolicy: Never
 restartPolicy: Never
---
kind: Service
apiVersion: v1
metadata:
  name: pod1-service
  namespace: k4y
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: pod1
---
kind: Service
apiVersion: v1
metadata:
  name: pod2-service
  namespace: k4y
spec:
  ports:
  - port: 80
```

```
    protocol: TCP
    targetPort: 8080
  selector:
    app: pod2
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: local-ingress
  namespace: k4y
spec:
  ingressClassName: nginx
  rules:
    - host: local-dev-env
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name:  pod1-service
                port:
                  number: 80
          - path: /
            pathType: Prefix
            backend:
              service:
                name:  pod2-service
                port:
                  number: 80
```

we need then to apply this manifest file

```
kubectl apply -f keep_for_yourself.yaml
```

this will create everything for us, and for verifying that is working we can just curl

```
$ curl http://local-dev-env/pod1
{"hello":"world 1"}
$ curl http://local-dev-env/pod2
{"hello":"world 2"}
```

Please note that in this article we used Python as base technology, in this article

're are explaining how to use go/golang (https://keepforyourself.com
.ubernetes/create-a-golang-microservice)

I hope this is opening your mind on the infinite possibilities you have at this stage! I also hope this helped you in some way and if so … share and let us grow!

If you want to know more about NGINX you can have a look at this book