# Practice M5: Advanced Concepts

For the purpose of this practice, we will assume that we are working on a machine with either a Windows 10/11 or any recent Linux distribution and there is a local virtualization solution (like VirtualBox, Hyper-V, VMware Workstation, etc.) installed

***Please note that long commands may be hard to read here. To handle this, you can copy them to a plain text editor first. This will allow you to see them correctly. Then you can use them as intended***

*For this practice we will assume that we are working on a three-node cluster (one control plane node and two worker nodes) with an arbitrary pod network plugin installed*

## Part 1: Static Pods and Multi-container Pods

Before we try the multi-container pods, let’s first check how to deal with static pods

### Static Pods

Static pod creation in terms of configuration or manifest doesn’t differ than creating a regular or standard pod

The only difference is how we send the manifest to the cluster

Here, with static pods, instead of passing it via the **API server** by using for example the **kubectl** tool, we save it to a special folder

By special, we mean a one which to be dedicated for this and to be registered in and monitored by the **kubelet** service

One way to do this is to alter its configuration and restart it

Here, we will use another approach, instead

First, log on to the control plane node

Then, let’s see some details about the **kubelet** process

**ps ax | grep /usr/bin/kubelet**

We may notice that it is reading different parts of its configuration from different files

Let’s check the contents of the **/var/lib/kubelet/config.yaml** file

**cat /var/lib/kubelet/config.yaml**

There are some interesting settings here, but we are interested in this row

**staticPodPath: /etc/kubernetes/manifests**

According to it, should we want to create a static pod, we must place its manifest into the stated folder

We should keep in mind that this will spin up the pod on the node in which folder we stored the manifest

Should we want the pod to run on another node, then we must save it in its special folder

Now, let’s see if there are any files in the folder

**ls -al /etc/kubernetes/manifests**

Wow, four components of the control plane are in fact running as static pods *(at least in a cluster, created by* ***kubeadm****)*

We can even check the manifest of the **etcd** database for example

**cat /etc/kubernetes/manifests/etcd.yaml**

Okay, enough exploring. Let’s test with our own manifest

Check this (**part1/1-static-pod.yaml**) manifest

apiVersion: v1

kind: Pod

metadata:

  name: static-pod

  labels:

    app: static-pod

spec:

  containers:

  - image: alpine

    name: main

    command: ["sleep"]

    args: ["1d"]

Copy it in the special folder and wait a few seconds. Then execute

**kubectl get pods -o wide**

Two things should grab our attention

First, the pod ran no matter that we are working on control plane node

Second, the name of the pod is not like usual. It is shorter and has the name of the node as a suffix

Now, let’s try to delete the pod using the usual approach

**kubectl delete pod static-pod-k8s1**

Then, check again

**kubectl get pods -o wide**

Ha, the pod is still there. Depending on how quick we executed the command, we may catch it in a different status but in any case, after a few seconds it will transition to running status

So, how we can delete it then? Perhaps via the container runtime (**Docker** in our case)?

Let’s try it but first we must find its container **ID**

**docker container ps | grep static-pod**

Now, we can delete it with

**docker container rm <container-id> --force**

Check again quickly

**docker container ps | grep static-pod**

No, just the special pause container is there

Check again. Now, our container is also there

If we check with **kubectl**

**kubectl get pods -o wide**

We will notice that if was just restarted *(it is because we killed the main container)*

Okay, how to get rid of this?

We must delete the manifest from the special folder where we copied it earlier

**rm /etc/kubernetes/manifests/1-static-pod.yaml**

Now, execute

**kubectl get pods -o wide -w**

And watch. After a while, the pod will be gone

Press **Ctrl+C** to return back to the terminal

Now, should we want to deploy it back, we just have to copy the manifest again. Skip it 😉

### Multi-container Pods

Static pods are fun but multi-container pods are even bigger fun

Let’s test two patterns – **sidecar** and **adapter**

#### Sidecar

Our first experiment will be with the sidecar pattern

Here, we will have the main container (a simple web application) and the sidecar container which will generate data consumed by the main container

Check this (**part1/2-sidecar.yaml**) manifest

apiVersion: apps/v1

kind: Deployment

metadata:

  name: sidecar

spec:

  replicas: 3

  selector:

    matchLabels:

      app: sidecar

  minReadySeconds: 15

  strategy:

    type: RollingUpdate

    rollingUpdate:

      maxUnavailable: 1

      maxSurge: 1

  template:

    metadata:

      labels:

        app: sidecar

    spec:

      containers:

      - name: cont-main

        image: shekeriev/k8s-appb

        imagePullPolicy: Always

        volumeMounts:

        - name: data

          mountPath: /var/www/html/data

        ports:

        - containerPort: 80

      - name: cont-sidecar

        image: alpine

        volumeMounts:

        - name: data

          mountPath: /data

        command: ["/bin/sh", "-c"]

        args:

          - while true; do

              date >> /data/generated-data.txt;

              sleep 10;

            done

      volumes:

      - name: data

        emptyDir: {}

---

apiVersion: v1

kind: Service

metadata:

  name: sidecar

  labels:

    app: sidecar

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: 30001

    protocol: TCP

  selector:

    app: sidecar

Pay attention to the common volume and how it is mounted to both containers

Send it to the cluster

**kubectl apply -f 2-sidecar.yaml**

And watch how it is progressing

**kubectl get pods -o wide -w**

Pay attention to the **0/2** value in the **READY** column

Once, all pods are running, press **Ctrl+C** to return to the terminal

Check all accompanying resources

**kubectl get pods,svc -o wide**

Open a browser and navigate to http://<control-plane>:<node-port>

A simple application should appear. Refresh a few times to check if the data generated by the sidecar container are coming and displayed

Check detailed information about one of the pods

**kubectl describe pod/sidecar-<identifier>**

Pay attention to the **Containers** section. There we can see the two containers and their settings

Should you want to establish a session to one of the containers, for example the sidecar, you must change a bit the exec command you used to use to something like this

**kubectl exec -it pod/sidecar-<identifier> -c cont-sidecar -- sh**

If we do not mind to which pod part of the deployment we will establish a session, we may change the command to the following

**kubectl exec -it deploy/sidecar -c cont-sidecar -- sh**

Browse the filesystem

Then close the session

Kill the main container

**kubectl exec -it pod/sidecar-<identifier> -c cont-main -- kill 1**

Quickly check the pods

**kubectl get pods**

You may catch it in a **NotReady** status

After a while, it will return to **Running** status

Remove all the traces by executing

**kubectl delete -f 2-sidecar.yaml**

#### Adapter

Next, we will simulate the adapter pattern with the following (**part1/3-adapter.yaml**) manifest

apiVersion: v1

kind: Pod

metadata:

  name: adapter

spec:

  containers:

  - name: cont-main

    image: alpine

    volumeMounts:

    - name: log

      mountPath: /var/log

    command: ["/bin/sh", "-c"]

    args:

      - while true; do

          echo $(date +'%Y-%m-%d %H:%M:%S') $(uname) OP$(tr -cd 0-1 </dev/urandom | head -c 1) $(tr -cd a-z </dev/urandom | head -c 5).html RE$(tr -cd 0-1 </dev/urandom | head -c 1) >> /var/log/app.log;

          sleep 3;

        done

  - name: cont-adapter

    image: alpine

    volumeMounts:

    - name: log

      mountPath: /var/log

    command: ["/bin/sh", "-c"]

    args:

      - tail -f /var/log/app.log | sed -e 's/^/MSG:/' -e 's/OP0/GET/' -e 's/OP1/SET/' -e 's/RE0/OK/' -e 's/RE1/ER/' > /var/log/out.log

  volumes:

  - name: log

    emptyDir: {}

Pay attention to both **args** sections

The first one is generating the log content and the second one is transforming it to match our imaginary requirements

Send it to the cluster

**kubectl apply -f 3-adapter.yaml**

Wait a bit until the pod is in **Running** state

**kubectl get pods**

Then check the source file in the main container

**kubectl exec -it adapter -c cont-main -- cat /var/log/app.log**

And then the transformed file in the adapter container

**kubectl exec -it adapter -c cont-adapter -- cat /var/log/out.log**

We may not see all messages in the output file initially but after a while they will appear

Now, remove the pod with

**kubectl delete -f 3-adapter.yaml**

### Init Containers

Let’s try a scenario with an init container

Check this (**part1/4-init-container.yaml**) manifest

apiVersion: v1

kind: Pod

metadata:

  name: pod-init

  labels:

    app: pod-init

spec:

  containers:

  - name: cont-main

    image: nginx

    ports:

    - containerPort: 80

    volumeMounts:

    - name: data

      mountPath: /usr/share/nginx/html

  initContainers:

  - name: cont-init

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - for i in $(seq 1 5); do

          echo $(date +'%Y-%m-%d %H:%M:%S') '<br />' >> /data/index.html;

          sleep 5;

        done

    volumeMounts:

    - name: data

      mountPath: /data

  volumes:

  - name: data

    emptyDir: {}

---

apiVersion: v1

kind: Service

metadata:

  name: svc-init

  labels:

    app: svc-init

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: 30001

    protocol: TCP

  selector:

    app: pod-init

Pay attention to the separate **initContainers** section. Considering the name, there could be more than one here

Send the file to the cluster

**kubectl apply -f 4-init-container.yaml**

Check the resources

**kubectl get pods,svc**

Watch the creation process

**kubectl get pods -w**

Once up and running, press **Ctrl+C**

Pay attention to the **READY** column. Here is just **/1** and not **/2** as with the other two (**sidecar** and **adapter**) patterns. Why?

Describe the pod

**kubectl describe pod pod-init**

Pay attention to the status of the init container (section **Init Containers**)

Open a browser and navigate to http://<control-plane>:<node-port>

Clean up by executing

**kubectl delete -f 4-init-container.yaml**

## Part 2: Autoscalling and Scheduling. Daemon Sets and Jobs

Let’s first start with the autoscaling experiments

### Autoscalling

We will test horizontal autoscaling

Before continuing further ensure that **metrics server** is up and running

If you are using **Minikube**, then you must enable the corresponding addon

**minikube addons enable metrics-server**

On a **custom made/standard Kubernetes** cluster install it by downloading the manifest

**wget https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml -O metrics-server.yaml**

Then open it and on line **#137** add the following

**--kubelet-insecure-tls**

Save and close the file

Use it to install the **metrics server**

**kubectl apply -f metrics-server.yaml**

More information about the metrics server can be found here:

<https://github.com/kubernetes-sigs/metrics-server>

Examine the manifest **part2/1-auto-scale.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: auto-scale-deploy

spec:

  replicas: 3

  selector:

    matchLabels:

      app: auto-scale

  template:

    metadata:

      labels:

        app: auto-scale

    spec:

      containers:

      - name: auto-scale-container

        image: shekeriev/terraform-docker

        ports:

        - containerPort: 80

        resources:

          requests:

            cpu: 100m

---

apiVersion: v1

kind: Service

metadata:

  name: auto-scale-svc

  labels:

    app: auto-scale

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: 30001

    protocol: TCP

  selector:

    app: auto-scale

And create the resources

**kubectl apply -f 1-auto-scale.yaml**

Check that all replicas are there by running

**kubectl get pods**

Once they are up and running, create an auto scale rule from this (**part2/1-auto-scale-hpa.yaml**) manifest

apiVersion: autoscaling/v2beta2

kind: HorizontalPodAutoscaler

metadata:

  name: auto-scale-deploy

spec:

  maxReplicas: 5

  minReplicas: 1

  scaleTargetRef:

    apiVersion: apps/v1

    kind: Deployment

    name: auto-scale-deploy

  metrics:

  - type: Resource

    resource:

      name: cpu

      target:

        type: Utilization

        averageUtilization: 10

**kubectl apply -f 1-auto-scale-hpa.yaml**

Ask for more information

**kubectl get horizontalpodautoscalers auto-scale-deploy**

Wait a few minutes for the metrics to be collected and ask again:

**kubectl get horizontalpodautoscalers auto-scale-deploy -o yaml**

After a few minutes (at least 5) the system should scale down our deployment to one replica

We can confirm this with

**kubectl get pods -o wide**

**kubectl get deployments**

Now we can open second terminal in order to monitor the scaling process (if running on Linux/macOS):

**watch -n 1 kubectl get hpa,deployment**

In the first terminal, we are going to simulate workload to trigger scale up:

**kubectl run -it --rm --restart=Never load-generator --image=busybox -- sh -c "while true; do wget -O - -q http://auto-scale-svc.default; done"**

If we switch to the other terminal, we can see the scale up process in action

Return to the first terminal and press **Ctrl+C** to stop the load generator pod

Check current quantity of replicas

**kubectl get deployments**

The scale down process will take some time (approx. 5 minutes). We won’t wait for it

Close the second terminal

Now, we can delete all resources

**kubectl delete -f 1-auto-scale-hpa.yaml**

**kubectl delete -f 1-auto-scale.yaml**

Disable the **metrics add on** should you want to (on **Minikube**):

**minikube addons disable metrics-server**

Or on a **standard Kubernetes** cluster, uninstall the **metrics server** should you want to

**kubectl delete -f metrics-server.yaml**

### Scheduling

We can influence the scheduling decision making process

One way of doing this by using **taints** and **tolerations** we will see now

Let’s first see if there are any existing taints in our cluster by executing this

**kubectl get nodes --show-labels**

Hm, no, we won’t see them using this approach. Let’s instead try to describe the first node

**kubectl describe node k8s1**

Pay attention to the **Taints** section

Now, try to get them for all nodes

**kubectl describe node | grep Taints**

No, the other two nodes don’t have any taints for now

So, this is the reason why there aren’t any user scheduled pods there (on the control plane node(s))

If we remove the taint, we should be able to schedule our pods there as well

This is done with *(skip it for now)*

**kubectl taint nodes k8s1 node-role.kubernetes.io/master:NoSchedule-**

But wait, how the system pods are scheduled on the control plane nodes then?

The answer is – they have **tolerations** for the **taints** of the control plane nodes. Let’s check

First, let’s see the list of some of those pods

**kubectl get pods -n kube-system -o wide**

Now, describe for example, the **etcd** pod

**kubectl describe pod etcd-k8s1 -n kube-system**

Hm, there aren’t any matching tolerations and yet the pod is running here. Why? *(Perhaps, because it is a static pod)*

Let’s check one of the **coredns** pods

**kubectl describe pod coredns-<identifier> -n kube-system**

Ha, finally there is the toleration we are looking for

Okay, let’s try this feature

Add a taint to one of the other two nodes

**kubectl taint node k8s2 demo-taint=nomorework:NoSchedule**

Check the situation with the **taints** again

**kubectl describe node | grep Taints**

Okay, now let’s spin a new deployment from this (**part2/2-schedule.yaml**) manifest

apiVersion: apps/v1

kind: Deployment

metadata:

  name: schedule-deploy

spec:

  replicas: 3

  selector:

    matchLabels:

      app: schedule

  template:

    metadata:

      labels:

        app: schedule

    spec:

      containers:

      - name: schedule-container

        image: shekeriev/terraform-docker

        ports:

        - containerPort: 80

        resources:

          requests:

            cpu: 100m

Send it to the cluster

**kubectl apply -f 2-schedule.yaml**

And check the distribution of the pods

**kubectl get pods -o wide**

All went to the third node (k8s3). Why? *(Perhaps, this is because of the taint)*

Now, delete the deployment

**kubectl delete -f 2-schedule.yaml**

And spin a new version (**part2/2-schedule-toleration.yaml**) with the following section added to the end

      tolerations:

      - key: demo-taint

        operator: Equal

        value: nomorework

        effect: NoSchedule

Send it to the cluster

**kubectl apply -f 2-schedule-toleration.yaml**

And check the distribution of the pods again

**kubectl get pods -o wide**

Now, there should be pods on the second node as well. Why? *(Perhaps, because of the toleration)*

Let’s clean a bit. First, remove the **taint**

**kubectl taint node k8s2 demo-taint-**

Then the deployment as well

**kubectl delete -f 2-schedule-toleration.yaml**

Now, in the next section, we will see another way of influencing the scheduling decision making process

### Daemon Sets

**Daemon Sets** are like the **Deployments**, **Replication Controllers** and **Replica Sets**

There is one important difference though – their workload goes to every node or only to specific nodes and with only one copy, so no multiple replicas spread across the cluster

Let’s check the manifest (**part-2/3-daemon-set.yaml**) file

apiVersion: apps/v1

kind: DaemonSet

metadata:

  name: daemon-set

spec:

  selector:

    matchLabels:

      app: daemon-set

  template:

    metadata:

      labels:

        app: daemon-set

    spec:

      nodeSelector:

        disk: samsung

      containers:

      - name: main

        image: shekeriev/k8s-appa:v1

        ports:

        - containerPort: 80

Pay attention to the **nodeSelector** block

Create the daemon set

**kubectl apply -f 3-daemon-set.yaml**

Check what has been created

**kubectl get ds**

Hm, nothing. Why?

Let’s ask for the list of the running pods

**kubectl get pods**

None. Strange, isn’t it?

Check the available nodes

**kubectl get nodes**

Information is too sparse. Let’s show the labels as well

**kubectl get nodes --show-labels**

Okay, if we remember correctly, in the manifest we set a node selector to look for **disk** of type **samsung**

It appears that none of our nodes has this key-value pair. So that is why nothing is scheduled yet

Let’s correct this and set a label of the node (for example the **second node**, or **k8s2**)

**kubectl label node k8s2 disk=samsung**

Now get the list of running pods

**kubectl get pods -o wide**

Ha, our pod is finally there

And then, some information about the daemon set

**kubectl get ds**

Let’s add the same label on the third node

**kubectl label node k8s3 disk=samsung**

And see how the things change

**kubectl get pods -o wide**

**kubectl get ds**

Now, we have two pods – one on every node matching the node selector

Let’s change the label of one of them (for example, node 2)

**kubectl label node k8s2 disk=wdc --overwrite**

And see what will happen

**kubectl get pods -o wide**

**kubectl get ds**

One of the pods (the one scheduled on node 2) is gone

Quite interesting feature 😊

Let’s clean up

**kubectl delete -f 3-daemon-set.yaml**

### Jobs

As we already know, there are two types of jobs. So, let’s explore them

#### Jobs

There are situations in which we need to run tasks that start, do something, and then finish

This is covered by a special object type – **Job**

Let’s check the following (**part2/4-batch-job.yaml**) manifest

apiVersion: batch/v1

kind: Job

metadata:

  name: batch-job

spec:

  template:

    metadata:

      labels:

        app: batch-job

    spec:

      restartPolicy: OnFailure

      containers:

      - name: main

        image: shekeriev/sleeper

This will launch a pod that will sleep for 60 seconds

Start the job

**kubectl apply -f 4-batch-job.yaml**

Get information about the job

**kubectl get jobs**

**kubectl get jobs -o wide**

Pay attention to the **COMPLETIONS** column

Check the pods

**kubectl get pods**

Get detailed information about the job

**kubectl describe job batch-job**

Pay attention to the **Parallelism**, **Completions**, and **Pod Statuses** fields

Check detailed information about our pod

**kubectl describe pod batch-job**

Depending on how long it took us to reach this point, the pod may already be in in **Terminated** state and the **Reason** will be because it has **Completed** its task *(to sleep for 60 seconds)*

Get again the info about the jobs

**kubectl get jobs**

Now the **COMPLETIONS** shows **1/1**

Delete the job (this will delete the pod as well)

**kubectl delete -f 4-batch-job.yaml**

We can run a job more than once. This can be done either in a sequence (serial) or in parallel

Let’s start with the serial one. Examine the next (**part2/4-batch-job-serial.yaml**) manifest

apiVersion: batch/v1

kind: Job

metadata:

  name: batch-job-serial

spec:

  completions: 3

  template:

    metadata:

      labels:

        app: batch-job

    spec:

      containers:

      - name: main

        image: shekeriev/sleeper

      restartPolicy: Never

Execute it

**kubectl apply -f 4-batch-job-serial.yaml**

Check the results

**kubectl get jobs**

Pay attention to the **COMPLETIONS** column. It shows that we are expecting **three** executions in total

Next, we can ask for a detailed information

**kubectl describe job batch-job-serial**

And finally, get the list of pods

**kubectl get pods**

We must repeat the above commands a few times in order to see the progress

Once done, we may delete the job

**kubectl delete -f 4-batch-job-serial.yaml**

It is time to test the parallel option as well

Examine the next manifest (**part2/4-batch-job-parallel.yaml**) file

apiVersion: batch/v1

kind: Job

metadata:

  name: batch-job-parallel

spec:

  completions: 4

  parallelism: 2

  template:

    metadata:

      labels:

        app: batch-job

    spec:

      containers:

      - name: main

        image: shekeriev/sleeper

      restartPolicy: Never

Pay attention to the **completions** and **parallelism** fields. We are expecting **four executions** and **two in parallel**

Execute it

**kubectl apply -f 4-batch-job-parallel.yaml**

Check the results

**kubectl get jobs**

**kubectl describe job batch-job-parallel**

**kubectl get pods -o wide**

We can see that we have two **pods running simultaneously** *(as expected)*

*Please note that they may be scheduled on* ***different nodes*** *or on* ***one and the same node***

Once the job is complete, we can delete it and all related objects with the usual command

**kubectl delete -f 4-batch-job-parallel.yaml**

#### Cron Jobs

There may be a need to execute job not once but on a schedule

This is solved by the **Cron Job** resource type

Examine the manifest (**part2/4-batch-job-cron.yaml**) file

apiVersion: batch/v1

kind: CronJob

metadata:

  name: batch-job-cron

spec:

  schedule: "\*/2 \* \* \* \*"

  jobTemplate:

    spec:

      template:

        metadata:

          labels:

            app: batch-job-cron

        spec:

          restartPolicy: OnFailure

          containers:

          - name: main

            image: shekeriev/sleeper

Pay attention to the **schedule** field. It will **run** **every two minutes**

Start the job

**kubectl apply -f 4-batch-job-cron.yaml**

Examine what happens

**kubectl get cronjobs**

**kubectl get cronjobs -o wide**

Check if there are any new pods created

**kubectl get pods -o wide**

No, we don’t have any yet. Why?

Repeat the check in let’s say one minute or so

Please note that it is not guaranteed where (on which node) the pod will be scheduled

Once done experimenting, delete the **Cron Job** *(it will delete the pods as well)*

**kubectl delete -f 4-batch-job-cron.yaml**

## Part 3: Ingress and Ingress Controllers

### Ingress Controller

We will try two ingress controllers – **NGINX** and **HAProxy**

#### NGINX

The detailed installation procedure can be found here:

<https://docs.nginx.com/nginx-ingress-controller/installation/installation-with-manifests/>

And here is the main repository:

<https://github.com/nginxinc/kubernetes-ingress>

First, we must make sure that we have a git client installed on the machine, which we will use for the procedure

Then, we must clone the repo locally

**git clone https://github.com/nginxinc/kubernetes-ingress/**

**cd kubernetes-ingress/deployments**

**git checkout v2.0.3**

Next, we must configure **RBAC**

Create the namespace and service account

**kubectl apply -f common/ns-and-sa.yaml**

Create the cluster role and cluster role binding

**kubectl apply -f rbac/rbac.yaml**

Then, we must create the needed common resources

First, we will create a secret that will hold the self-signed **TLS** certificate

**kubectl apply -f common/default-server-secret.yaml**

Next, the configuration map that may be used for **NGINX** customization

**kubectl apply -f common/nginx-config.yaml**

Then, we must create the ingress class resource

**kubectl apply -f common/ingress-class.yaml**

And a few more required custom resource definitions - for **VirtualServer**, **VirtualServerRoute**, **TransportServer** and **Policy**

**kubectl apply -f common/crds/k8s.nginx.org\_virtualservers.yaml**

**kubectl apply -f common/crds/k8s.nginx.org\_virtualserverroutes.yaml**

**kubectl apply -f common/crds/k8s.nginx.org\_transportservers.yaml**

**kubectl apply -f common/crds/k8s.nginx.org\_policies.yaml**

Now, we are ready to deploy the ingress controller

For this, we can use either **Deployment** (if we want to be in control and be able to change the number of replicas) or **DaemonSet** (if we want one controller per node or set of nodes)

Let’s go with the **Deployment** option

**kubectl apply -f deployment/nginx-ingress.yaml**

We can watch the installation process with

**kubectl get pods --namespace=nginx-ingress -w**

Press **Ctrl+C** when done

Now, let’s create a **NodePort** service to access the ingress controller

**kubectl create -f service/nodeport.yaml**

Check the service with

**kubectl get service -n nginx-ingress**

We are done here (for now) 😉

#### HAProxy

The main repository with any additional information can be found here:

<https://github.com/haproxytech/kubernetes-ingress>

With **HAProxy**, the installation procedure is simpler compared to **NGINX**

It is enough to execute this

**kubectl apply -f** [**https://raw.githubusercontent.com/haproxytech/kubernetes-ingress/master/deploy/haproxy-ingress.yaml**](https://raw.githubusercontent.com/haproxytech/kubernetes-ingress/master/deploy/haproxy-ingress.yaml)

Of course, should we want to customize it, we must clone the repository locally first

As we have two ingress controllers, we must create an ingress class for this one *(the* ***NGINX*** *one created one)*

Prepare a **haproxy-class.yaml** manifest with the following content

**apiVersion: networking.k8s.io/v1**

**kind: IngressClass**

**metadata:**

**name: haproxy**

**spec:**

**controller: haproxy-ingress.github.io/controller**

Save it and close it

Send it to the cluster

**kubectl apply -f haproxy-class.yaml**

Let’s see if have both classes

**kubectl get ingressclass**

We are done here (for now) 😉

### Ingress

Let’s test the following three scenarios with one of the two ingress controllers we installed

For example, let’s do it with the **NGINX** one

#### Single Service

Let’s add one pod and expose it via service with type **ClusterIP**

This can be done with this (**part3/pod-svc-1.yaml**) manifest

apiVersion: v1

kind: Pod

metadata:

  name: pod1

  labels:

    app: pod1

spec:

  containers:

  - image: shekeriev/k8s-environ

    name: main

    env:

    - name: TOPOLOGY

      value: "POD1 -> SERVICE1"

    - name: FOCUSON

      value: "TOPOLOGY"

---

apiVersion: v1

kind: Service

metadata:

  name: service1

spec:

  ports:

  - port: 80

    protocol: TCP

  selector:

    app: pod1

Once, we send it to the cluster, we can check that both resources are ready

**kubectl get pod,svc**

Then, we will create an ingress controller using this (**part3/1-nginx-single.yaml**) manifest

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: ingress-ctrl

spec:

  ingressClassName: nginx

  rules:

  - host: demo.lab

    http:

      paths:

      - path: /

        pathType: Prefix

        backend:

          service:

            name: service1

            port:

              number: 80

Sent it to the cluster

**kubectl apply -f 1-nginx-single.yaml**

Then we can check if the resource is ready

**kubectl get ingress**

And why not, describe it with

**kubectl describe ingress ingress-ctrl**

Get the **NodePort** of the ingress service

**kubectl get svc nginx-ingress -n nginx-ingress**

*For the* ***HAProxy****, you must execute*

***kubectl get svc haproxy-ingress -n haproxy-controller***

Open a browser tab and navigate to [http://demo.lab:<node-port](http://demo.lab:%3cnode-port)>

*Please note that you should have a record in your* ***hosts file*** *that matches* ***demo.lab*** *to the* ***IP address of the control plane node***

It is working! 😊

#### Single Service with Custom Path

Now, let’s try something else

Instead of the base **URL** let’s use a custom path

Check this (**part3/2-nginx-custom-path-a.yaml**) manifest

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: ingress-ctrl

spec:

  ingressClassName: nginx

  rules:

  - host: demo.lab

    http:

      paths:

      - path: /service1

        pathType: Prefix

        backend:

          service:

            name: service1

            port:

              number: 80

And send it to the cluster

**kubectl apply -f 2-nginx-custom-path-a.yaml**

It will overwrite the existing ingress controller

Now, check how it looks like

**kubectl describe ingress ingress-ctrl**

Pay attention to the **Rules** section

Open a browser tab and navigate to [http://demo.lab:<node-port>/service1](http://demo.lab:%3cnode-port%3e/service1)

Is it working? No. Why? *No idea, at least not yet*

Let’s ask for the logs of the pod that is behind the service

**kubectl logs pod1**

We will notice that the last message contains

**"GET /service1 HTTP/1.1" 404**

Which means that the ingress is sending **/service1** as **URL** to the service

Okay, but our service listens on **/** instead of **/service1**

We may address this by utilizing the so-called rewrite rules

Let’s check this (**part3/2-nginx-custom-path-b.yaml**) manifest

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: ingress-ctrl

  annotations:

    nginx.org/rewrites: "serviceName=service1 rewrite=/"

spec:

  ingressClassName: nginx

  rules:

  - host: demo.lab

    http:

      paths:

      - path: /service1

        pathType: Prefix

        backend:

          service:

            name: service1

            port:

              number: 80

And send it to the cluster (it will overwrite the existing ingress controller)

**kubectl apply -f 2-nginx-custom-path-b.yaml**

We may check again how it changed

**kubectl describe ingress ingress-ctrl**

Open a browser tab and navigate to [http://demo.lab:<node-port>/service1](http://demo.lab:%3cnode-port%3e/service1)

Is it working? Yes, it does 😊

#### Default Backend

Add two more resources from this (**pod-svc-d.yaml**) manifest

apiVersion: v1

kind: Pod

metadata:

  name: podd

  labels:

    app: podd

spec:

  containers:

  - image: shekeriev/k8s-environ

    name: main

    env:

    - name: TOPOLOGY

      value: "PODd -> SERVICEd (default backend)"

    - name: FOCUSON

      value: "TOPOLOGY"

---

apiVersion: v1

kind: Service

metadata:

  name: serviced

spec:

  ports:

  - port: 80

    protocol: TCP

  selector:

    app: podd

Then, let’s use another manifest (**part3/3-nginx-default-back.yaml**) with the following content

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: ingress-ctrl

  annotations:

    nginx.org/rewrites: "serviceName=service1 rewrite=/"

spec:

  ingressClassName: nginx

  defaultBackend:

    service:

      name: serviced

      port:

        number: 80

  rules:

  - host: demo.lab

    http:

      paths:

      - path: /service1

        pathType: Prefix

        backend:

          service:

            name: service1

            port:

              number: 80

Send it to the cluster (it will overwrite the existing one)

**kubectl apply -f 3-nginx-default-back.yaml**

And then check how the ingress controller changed

**kubectl describe ingress ingress-ctrl**

Note the **Default backend** section and the **Rules** section

Open a browser tab and navigate to [http://demo.lab:<node-port](http://demo.lab:%3cnode-port)>

Ha, it is working and showing different output

Now, check the previous URL - [http://demo.lab:<node-port>/service1](http://demo.lab:%3cnode-port%3e/service1)

Also working 😊

#### Fan Out

Let’s extend the setup by adding another pair of pod and service first (**part3/pod-svc-2.yaml**)

apiVersion: v1

kind: Pod

metadata:

  name: pod2

  labels:

    app: pod2

spec:

  containers:

  - image: shekeriev/k8s-environ

    name: main

    env:

    - name: TOPOLOGY

      value: "POD2 -> SERVICE2"

    - name: FOCUSON

      value: "TOPOLOGY"

---

apiVersion: v1

kind: Service

metadata:

  name: service2

spec:

  ports:

  - port: 80

    protocol: TCP

  selector:

    app: pod2

And then, change the ingress controller configuration (**part3/4-nginx-fan-out.yaml**) to match this

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: ingress-ctrl

  annotations:

    nginx.org/rewrites: "serviceName=service1 rewrite=/;serviceName=service2 rewrite=/"

spec:

  ingressClassName: nginx

  defaultBackend:

    service:

      name: serviced

      port:

        number: 80

  rules:

  - host: demo.lab

    http:

      paths:

      - path: /service1

        pathType: Prefix

        backend:

          service:

            name: service1

            port:

              number: 80

      - path: /service2

        pathType: Prefix

        backend:

          service:

            name: service2

            port:

              number: 80

Once sent to the cluster, check how the ingress controller has changed

**kubectl describe ingress ingress-ctrl**

Pay attention to the **Rules** section

Now, test all three **URLs**

Open a browser tab and navigate to [http://demo.lab:<node-port](http://demo.lab:%3cnode-port)>

Now, check the **service1** **URL** - [http://demo.lab:<node-port>/service1](http://demo.lab:%3cnode-port%3e/service1)

And finally, check the **service2** **URL** - [http://demo.lab:<node-port>/service2](http://demo.lab:%3cnode-port%3e/service2)

All three are working 😊

#### Name Based Virtual Hosting

Let’s try one more way of using the ingress controller

Check this (**part3/5-nginx-name-vhost.yaml**) manifest

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: ingress-ctrl

spec:

  ingressClassName: nginx

  rules:

  - host: demo.lab

    http:

      paths:

      - pathType: Prefix

        path: "/"

        backend:

          service:

            name: service1

            port:

              number: 80

  - host: awesome.lab

    http:

      paths:

      - pathType: Prefix

        path: "/"

        backend:

          service:

            name: service2

            port:

              number: 80

Send it to the cluster (it will overwrite the current ingress controller)

**kubectl apply -f 5-nginx-name-vhost.yaml**

Check how the ingress controller has changed

**kubectl describe ingress ingress-ctrl**

Pay attention to the **Rules** section

Before testing, make sure that you have records for both **demo.lab** and **awesome.lab** in your **hosts** file

Then, open a browser and navigate to [http://demo.lab:<node-port](http://demo.lab:%3cnode-port)>

It should work and show the contents of **service1**

Now, open another browser tab and navigate to [http://awesome.lab:<node-port](http://awesome.lab:%3cnode-port)>

It should work also and show the contents of **service2**

#### *Try Other Options*

*By now, we should have a good understanding of how the ingress is working*

*Should we want, we can try now with the other one, we installed –* ***HAProxy*** *(there is a separate set of files)*

*Don’t forget to remove the ingress controller resource first*

### Clean Up

First, we must delete the application artifacts

**kubectl delete pods podd pod1 pod2**

**kubectl delete svc serviced service1 service2**

**kubectl delete ingress ingress-ctrl**

#### NGINX

Delete the whole namespace

**kubectl delete namespace nginx-ingress**

Then the cluster role and cluster role binding

**kubectl delete clusterrolebinding nginx-ingress**

**kubectl delete clusterrole nginx-ingress**

And finally, all custom resource definitions

Navigate back to the cloned repository and execute

**kubectl delete -f common/crds/**

#### HAProxy

Delete everything at once

**kubectl delete -f https://raw.githubusercontent.com/haproxytech/kubernetes-ingress/master/deploy/haproxy-ingress.yaml**