# Solution M5: Advanced Concepts

For this challenge we will assume that we are working as the **root** user on the control plane node of a three node **Kubernetes** cluster. The pod network plugin is of no importance, so it can be any of the available. We will execute all tasks in order but keep in mind that every other task expects that the artefacts from the previous have been deleted

# Task 1

**Challenge:**

**Init Containers**

* 1. Create a set of two **init containers** and **one app container** *(in fact modify/extend the example shown during the practice)*
  2. The **first init container** should generate the following two lines with **10 seconds** delay

***dd-mm-yyyy hh:mi:ss => begin initialization …***

***dd-mm-yyyy hh:mi:ss => … done***

*Please note that the dd-mm-yyyy hh:mi:ss should reflect the actual time the event is taking place*

* 1. The **second init container** should add one more line like the following

***dd-mm-yyyy hh:mi:ss => launching the application …***

*Please note that the dd-mm-yyyy hh:mi:ss should reflect the actual time the event is taking place*

* 1. The **app container** should be **nginx** based and should **display the three lines** generated by the init containers instead of the **default index page**

**Solution:**

Using the manifest (**part1/4-init-container.yaml**) from the practice, we can build the one (**task1/pod-svc.yaml**) that will give the solution to this task. It will look like

apiVersion: v1

kind: Pod

metadata:

  name: pod-init

  labels:

    app: pod-init

spec:

  containers:

  - name: cont-app-main

    image: nginx

    ports:

    - containerPort: 80

    volumeMounts:

    - name: data

      mountPath: /usr/share/nginx/html

  initContainers:

  - name: cont-init-1

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

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

        sleep 10;

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

    volumeMounts:

    - name: data

      mountPath: /data

  - name: cont-init-2

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - echo -e $(date +'%Y-%m-%d %H:%M:%S') '=> launching the application ...<br />\n' >> /data/index.html;

    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

Deploy the manifest to the cluster

**kubectl apply -f pod-svc.yaml**

And open a browser tab and navigate to **http://<cluster-ip>:30001**

Our application should be there up and running

# Task 2

**Challenge:**

**Ingress** and **TLS**

* 1. Using either **NGINX** or **HAProxy** *(the implementation should not differ significantly)* **ingress controller** try to modify/extend the **fan out** example shown in the practice to **handle TLS traffic**
  2. Note, that you will need to create a **self-signed certificate** and store it in a **secret** which then to be **used in the ingress**

**Solution:**

We will use the following files from the practice:

* Pod + service manifests (**part3/pod-svc1.yaml** and **part3/pod-svc2.yaml**)
* NGINX fan out example (**part3/4-nginx-fan-out.yaml**)

Of course, we will modify the later to match the requirements of this task

First, we should prepare the self-signed certificate

**openssl genrsa -out hw.key 4096**

**openssl req -new -x509 -key hw.key -out hw.crt -days 365 -subj /CN=demo.lab**

Then, we should create a secret to store the certificate

**kubectl create secret tls hw-sec --key=hw.key --cert=hw.crt**

Of course, we should have a record in our hosts file for the **demo.lab** domain that points to the control plane

Next, we must prepare a manifest (**nginx-fan-out.yaml**) for the ingress resource with the following content

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: ingress-ctrl

  annotations:

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

spec:

  ingressClassName: nginx

  tls:

  - hosts:

    - demo.lab

    secretName: hw-sec

  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

Deploy all manifests with

**kubectl apply -f pod-svc-1.yaml**

**kubectl apply -f pod-svc-2.yaml**

**kubectl apply -f nginx-fan-out.yaml**

Check that the ingress resource was deployed correctly

**kubectl describe ingress**

Check the node port that we should use

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

And then open a browser tab and navigate to

* [https://demo.lab:<https-node-port>/service1](https://demo.lab:%3chttps-node-port%3e/service1)
* [https://demo.lab:<https-node-port>/service2](https://demo.lab:%3chttps-node-port%3e/service2)

The HTTPS node port above is the one to which the 443 port is redirected

You should accept the warning for the security risk (we are using a self-signed certificate)

And voila, our setup is working 😊

# Task 3

**Challenge:**

Another **Ingress Controller**

* 1. Repeat the **fan out** example shown in the practice but with another **ingress controller of your choice** *(not* ***NGINX*** *or* ***HAProxy****)*

**Solution:**

Partial list of available ingress controllers can be seen here:

<https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>

For this solution, we will choose the **Contour** ingress controller (in fact it is not just an ingress controller but a HTTP proxy):

<https://projectcontour.io>

It is based on the **Envoy** proxy:

<https://www.envoyproxy.io/>

*Usually, the installation procedure is extremely simple. It is a matter of executing a single command (skip this)*

***kubectl apply -f https://projectcontour.io/quickstart/contour.yaml***

Instead, as we are working with on-premises cluster, we will download the manifest first

**wget https://projectcontour.io/quickstart/contour.yaml**

Then, open it and navigate to row **4944** (this should be start of the **envoy** service definition)

There, change the following:

* Row **4958**, from **externalTrafficPolicy: Local** to **externalTrafficPolicy: Cluster**
* Row **4970**, from **type: LoadBalancer** to **type: NodePort**

Save and close the file. Deploy it with

**kubectl apply -f contour.yaml**

After a while, we can check the status of the deployed pods and services with

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

*Next, if we have more than one ingress controller, we must create a class (skip this)*

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

*apiVersion: networking.k8s.io/v1*

*kind: IngressClass*

*metadata:*

*name: contour*

*spec:*

*controller: projectcontour.io/projectcontour/contour*

*And then deploy it with*

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

As we will see in a while, **Contour** provides more than just an ingress controller

Then, reuse the following manifests (**part3/pod-svc1.yaml** and **part3/pod-svc2.yaml**) from the practice and deploy them with

**kubectl apply -f pod-svc-1.yaml**

**kubectl apply -f pod-svc-2.yaml**

Finally, prepare a manifest, a variant of **part3/4-nginx-fan-out.yaml** used during the practice

Here, we will use the **HTTPProxy** object (which we can consider as an improved ingress controller) provided by **Contour**

Let’s create a manifest **contour-http-proxy.yaml** with the following content

apiVersion: networking.k8s.io/v1

apiVersion: projectcontour.io/v1

kind: HTTPProxy

metadata:

  name: contour-proxy

spec:

  virtualhost:

    fqdn: demo.lab

  routes:

    - conditions:

      - prefix: /service1

      services:

        - name: service1

          port: 80

      pathRewritePolicy:

        replacePrefix:

        - prefix: /service1

          replacement: /

    - conditions:

      - prefix: /service2

      services:

        - name: service2

          port: 80

      pathRewritePolicy:

        replacePrefix:

        - prefix: /service2

          replacement: /

As we can see, it is indeed an improved/extended version of what we did with the ingress controller in the fan out example

Deploy it with

**kubectl apply -f contour-http-proxy.yaml**

And check the resulting object with

**kubectl describe httpproxy**

Then, we can open a browser tab and navigate to

* Service 1 – http://<cluster-ip>:<http-port>/service1
* Service 1 – http://<cluster-ip>:<http-port>/service1

The **<cluster-ip>** can be either of control plane node or worker node address

The **<http-port>** we can see by executing

**kubectl get svc -n projectcontour**

And copying the appropriate value for the **envoy** service

That is, it. We made it 😊