Lab 11A. Pods

In this lab, you will learn how to launch applications using the basic deployment unit of Kubernetes, i.e. *pods*. This time, you are going to do it by writing declarative configs with *YAML* syntax.

Launching Pods with Kubernetes

Downloading Helper Code

To download the helper code for this course, switch to the workstation which has been configured with **`kubectl`** and clone the **`k8s-code`** repo:

```
git clone https://github.com/LFD254/k8s-code.git
```

This is a very important step. Do not miss it!

Launching Pods Manually

You can use **`kubectl run`** to launch a pod by specifying just the image.

For example, if you want to launch a pod for Redis, with an image **`redis:alpine`**, you should use the following command:

```
kubectl run redis  --image=redis
```

Kubernetes Resources and Writing YAML Specs

Each entity created with Kubernetes is a resource including pod, service, deployments, replication controller, etc. Resources can be defined as YAML or JSON. Here is the syntax to create a YAML specification:

AKMS > Resource Configs Specs

```
apiVersion: v1
kind:
metadata:
spec:
```

Use this Kubernetes API Reference Document to write the API specs. This is the most important reference while working with Kubernetes, so it's highly recommended that you bookmark it for the version of Kubernetes that you use.

To find the version of Kubernetes use the following command:

```
kubectl version -o yaml
```

To list the running pods:

```
kubectl get pods
```

To list API objects, use the following command:

```
kubectl api-resources
```

Writing Pod Spec

Let's now create the pod config by adding **kind** and **specs** to the schema given in the file **vote-pod.yaml** as follows:

Filename: **k8s-code/pods/vote-pod.yaml**

```
apiVersion:
kind: Pod
metadata:
spec:
```

Problem statement: Create a YAML spec to launch a pod with one container and to run a **vote** application, which matches the following specs:

```
pod:
 metadata:
    name: vote
    labels:
        app: python
        role: vote
        version: v1
spec
   containers:
```

```
    name: app
    image: schoolofdevops/vote:v1
```

Refer to the Kubernetes API Reference Pod v1 Core Document to find out the relevant properties of a pod and add those to the **vote-pod.yaml** provided in the supporting code repository.

Filename: **k8s-code/pods/vote-pod.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: vote
  labels:
    app: python
    role: vote
    version: v1
spec:
  containers:
    - name: app
      image: schoolofdevops/vote:v1
```

Use this example to refer to the pod spec.

Launching and Operating Pods

To launch a monitoring screen and see what's being launched, use the following command in a new terminal window where **kubectl** is configured:

```
watch kubectl get all
```

kubectl syntax:

```
kubectl
kubectl apply --help
kubectl apply -f FILE
```

To do a dry run before applying use the following command:

```
kubectl apply -f vote-pod.yaml --dry-run=client
```

To *launch* a pod using configs above, remove dry run options and run:

```
kubectl apply -f vote-pod.yaml
```

To *view* pods:

```
kubectl get pods
kubectl get po
kubectl get pods -o wide
kubectl get pods --show-labels
kubectl get pods -l "role=vote,version=v1"
kubectl get pods vote
```

To get detailed info:

```
kubectl describe pods vote
```

To *operate* the pod:

```
kubectl logs vote
kubectl exec -it vote sh
```

Run the following commands inside the container in a pod after running **`exec`** command:

```
ifconfig
cat /etc/issue
hostname
cat /proc/cpuinfo
ps aux
```

Use **`^d`** or **`exit`** to log out.

Adding Volume for Data Persistence

Let's create a pod for the database and attach a volume to it. To achieve this we will need to:

- Create a **volumes** definition
- Attach volume to container using **`VolumeMounts`** property

Localhost volumes are of two types:

- **`emptyDir`**
- **`hostPath`**

We will select hostPath (to learn more about **`hostPath`** read Kubernetes Documentation).

File: **`db-pod.yaml`**

```
apiVersion: v1
kind: Pod
metadata:
  name: db
```

```
  labels:
    app: postgres
    role: database
    tier: back
spec:
  containers:
    - name: db
      image: postgres:9.4
      ports:
        - containerPort: 5432
      volumeMounts:
      - name: db-data
        mountPath: /var/lib/postgresql/data
  volumes:
  - name: db-data
    hostPath:
      path: /var/lib/pgdata
      type: DirectoryOrCreate
```

To create this pod:

```
kubectl apply -f db-pod.yaml
kubectl get pods
```

At this time, you may see that the pod is in **`CrashLoopBackOff`** state. Try debugging the issue by checking the logs and resolving it before proceeding.

Once resolved, confirm that the pod is in a running state:

```
kubectl get pods
kubectl describe pod db
kubectl get events
```

Examine **`/var/lib/pgdata`** on the node on which it's scheduled by running **`kubectl get pods -o wide to find the node`**. Check if the directory has been created and if the data is present.

Creating Multi-container Pods (Sidecar Example)

File: **`multicontainerpod.yaml`**

```
apiVersion: v1
kind: Pod
metadata:
  name: web
```

```
  labels:
    tier: front
    app: nginx
    role: ui
spec:
  containers:
    - name: nginx
      image: nginx:stable-alpine
      ports:
        - containerPort: 80
          protocol: TCP
      volumeMounts:
        - name: data
          mountPath: /var/www/html-sample-app

    - name: sync
      image: schoolofdevops/sync:v2
      volumeMounts:
        - name: data
          mountPath: /var/www/app

  volumes:
    - name: data
      emptyDir: {}
```

To create this pod:

```
kubectl apply -f multi_container_pod.yaml
```

Check status:

```
root@kube-01:~# kubectl get pods
NAME       READY     STATUS              RESTARTS   AGE
web        0/2       ContainerCreating   0          7s
vote       1/1       Running             0          3m
```

Check logs and log in:

```
kubectl logs  web  -c sync
kubectl logs  web  -c nginx

kubectl exec -it web  sh  -c nginx
kubectl exec -it web  sh  -c sync
```

Using the following commands, observe which elements are common and which are isolated in two containers running inside the same pod.

Shared:
hostname
ifconfig

Isolated:

cat /etc/issue
ps aux
df -h

Adding Resource Requests and Limits

We can control the amount of resources requested and also put a limit on how many resources a container in a pod can take on. This can be done by adding spec to the existing pod as presented below. Refer to the Kubernetes Documentation, *"Managing Resources for Containers"*, to learn more.

Filename: **vote-pod.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: vote
  labels:
    app: python
    role: vote
    version: v1
spec:
  containers:
    - name: app
      image: schoolofdevops/vote:v1
      resources:
        requests:
          memory: "64Mi"
          cpu: "50m"
        limits:
          memory: "128Mi"
          cpu: "250m"
```

Let's now apply the changes:

```
kubectl apply -f vote-pod.yaml
```

If you already have a **vote** pod running, you may see an output similar to this presented below:

```
The Pod "vote" is invalid: spec: Forbidden: pod updates may not change
fields other than `spec.containers[*].image`,
`spec.initContainers[*].image`, `spec.activeDeadlineSeconds` or
`spec.tolerations` (only additions to existing tolerations)
{"Volumes":[{"Name":"default-token-snbj4","HostPath":null,"EmptyDir":n
ull,"GCEPersistentDisk":null,"AWSElasticBlockStore":null,"GitRepo":nul
l,"Secret":{"SecretName":"default-token-snbj4","Items":null,"DefaultMo
de":420,"Optional":null},"NFS":null,"ISCSI":null,"Glusterfs":null,"Per
sistentVolumeClaim":null,"RBD":null,"Quobyte":null,"FlexVolume":null,"
Cinder":null,"CephFS":null,"Flocker":null,"DownwardAPI":null,"FC":null
,"AzureFile":null,"ConfigMap":null,"VsphereVolume":null,"AzureDisk":nu
ll,"PhotonPersistentDisk":null,"Projected":null,"PortworxVolume":null,
"ScaleIO":null,"StorageOS":null}],"InitContainers":null,"Containers":[
{"Name":"app","Image":"schoolofdevops/vote:v1","Command":null,"Args":n
ull,"WorkingDir":"","Ports":null,"EnvFrom":null,"Env":null,"Resources"
:{"Limits":
....
...
```

From the above output, it's clear that not all the fields are mutable (except for a few, e.g labels). Container-based deployments primarily follow the concept of **immutable deployments**. So to bring your change into effect, you need to re-create the pod:

```
kubectl delete pod vote
kubectl apply -f vote-pod.yaml
kubectl describe pod vote
```

Based on the output of the **`describe`** command, you can confirm that the resource constraints you added are in place.

Now:

- Define the value of **`cpu.request`** > **`cpu.limit`**. Try to apply the changes made to the manifests, and observe.
- Define the values for **`memory.request`** and **`memory.limit`** that are higher than the total system memory. Apply the changes made to the manifest and observe the deployment and pods.

Deleting Pods

Now that you are done experimenting with a pod, delete it with the following command:

```
kubectl delete pod vote web db
kubectl get pods
```

Additional Resources

- *"Types of Volumes"*
- *"Assigning Pods to Nodes"*

Summary

In this lab, you learned how to create, manage and work with the most fundamental Kubernetes primitive: pod. You have also begun your journey writing YAML manifests, which is an essential skill to master Kubernetes. You also learned how to use kubectl to create, update and manage resources. And finally, you explored the most common issues with the pods and how to troubleshoot those.