Kubernetes

Pods: Basics

In this lab we will explore the nature of Kubernetes pods and how to work with them.

Like individual application containers, pods are considered to be relatively ephemeral rather than durable entities. Pods are scheduled to nodes and remain there until termination (according to restart policy) or deletion. When a node dies, the pods scheduled to that node are deleted. Specific pods are never moved to new nodes; instead, they must be replaced by running fresh copies of the images on another node.

Kubernetes supports declarative YAML or JSON configuration files. Often times config files are preferable to imperative commands, since they can be checked into version control and changes to the files can be code reviewed, producing a more robust, reliable and CI/CD or GitOps friendly system. They can also save a lot of typing if you would like to deploy complex pods or entire applications.

0. Prerequisites

If you already know how to login to your lab system and Docker and Kubernetes are installed and configured you can skip this section and jump directly to `1.` . Otherwise, follow these instructions

0.1 Login to your lab system

Login to your machine using an ssh client:

```
@laptop:~$ chmod 400 key.pem

@laptop:~$ ssh -i <class SSH Key>.pem ubuntu@<your VM IP>

Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-1031-aws x86_64)

...

~$
```

0.2. Install and configure Docker and Kubernetes

For this lab you will need a Kubernetes cluster. We have prepared a quick installation script that expedites the process of setting up your Lab VM for Kubernetes by performing the following operations:

- Downloading and Installing Docker
- Enabling the apt package manager to retrieve the Kubernetes binaries
- Bootstrapping a Kubernetes Cluster

- Configuring kubectl
- Allowing regular workloads to run on a single node

```
~$ wget -qO - https://raw.githubusercontent.com/RX-
M/classfiles/master/k8s.sh | sh

...

~$
```

We also add our account to the `docker` group to minimize use of `sudo`:

```
~$ sudo usermod -aG docker $(whoami)     # add yourself to the group

~$
```

Add tab completion permanently to your bash shell:

```
~$ echo "source <(kubectl completion bash)" >> ~/.bashrc
```

After updating your user groups and adding tab completion you will need to refresh your shell session. On the lab system the easiest approach is to logout at the command line:

```
~$ exit

logout
Connection to 55.55.55.55 closed.

@laptop:~$
```

Log back in as *ubuntu* with the SSH key:

```
@laptop:~$ ssh -i \path-to-ssh-key\key.pem ubuntu@<your VM IP>

...

~$
```

Great. Your current shell, and any new shell sessions, can now use the `docker` command without `sudo` and can make use of tab completion.

1. A simple pod

To begin our exploration, we'll create a basic Kubernetes pod from the command line. The easiest way to run a pod is using the `kubectl run` command. Try creating a simple Apache Web Server pod using the `kubectl run` subcommand as follows.

```
~$ kubectl run apache --image=docker.io/httpd:2.2

pod/apache created

~$
```

Now view the pod:

```
~$ kubectl get pod

NAME      READY   STATUS    RESTARTS   AGE
apache    1/1     Running   0          8s

~$
```

What happened here?

The `kubectl run` command takes a name, an image and an API object as parameters and it generates pod template including the image you specified.

The `run` subcommand syntax is as follows:

```
~$ kubectl run -h | grep COMMAND

  kubectl run NAME --image=image [--env="key=value"] [--port=port] [--dry-run=server|client] [--overrides=inline-json] [--command] -- [COMMAND] [args...] [options]

~$
```

- `--env` switch sets environment variables (just like the `docker run -e` switch)
- `--port` switch exposes ports for service mapping
- `--dry-run` switch (if set to client) allows you to submit the command without executing it to test the parameters
- `--overrides` switch can be used to provide json for fields that don't have flags
- `--command` switch will accept paths to binaries and arguments to pass to those binaries, note the `--` required for the syntax

To get more information about our pod use the `kubectl describe` subcommand:

```
~$ kubectl describe pod apache

Name:             apache
Namespace:        default
Priority:         0
Service Account:  default
Node:             ip-172-31-4-161/172.31.4.161
Start Time:       Thu, 18 Jan 2024 01:07:23 +0000
Labels:           run=apache
Annotations:      <none>
Status:           Pending
IP:
IPs:              <none>
Containers:
  apache:
    Container ID:
    Image:          docker.io/httpd:2.2
    Image ID:
    Port:           <none>
    Host Port:      <none>
    State:          Waiting
      Reason:       ContainerCreating
    Ready:          False
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-
blgw9 (ro)
Conditions:
  Type                        Status
  PodReadyToStartContainers   False
  Initialized                 True
  Ready                       False
  ContainersReady             False
  PodScheduled                True
Volumes:
  kube-api-access-blgw9:
    Type:                    Projected (a volume that contains injected
data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute
op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute
op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  11s   default-scheduler  Successfully assigned
```

```
default/apache to ip-172-31-4-161
  Normal  Pulling    10s   kubelet            Pulling image
"docker.io/httpd:2.2"

~$
```

Read through the *Events* reported for the pod.

You can see that Kubernetes used containerd to pull, create, and start the httpd image requested. You can also see which part of Kubernetes caused the event. For example the scheduler assigned the pod to the node and then the Kubelet on the assigned node starts the container.

Kubernetes also injects a standard set of environment variables into the containers of a pod providing the location of the cluster API service. Use `kubectl exec` to see the pod's environment variables. We'll go into further details about `kubectl exec` later.

```
~$ kubectl exec apache -- env

PATH=/usr/local/apache2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/bin
HOSTNAME=apache
HTTPD_PREFIX=/usr/local/apache2
HTTPD_VERSION=2.2.34
HTTPD_SHA256=e53183d5dfac5740d768b4c9bea193b1099f4b06b57e5f28d7caaf9ea7498
160
HTTPD_PATCHES=CVE-2017-9798-patch-2.2.patch
42c610f8a8f8d4d08664db6d9857120c2c252c9b388d56f238718854e6013e46 2.2.x-
mod_proxy-without-APR_HAS_THREADS.patch
beb66a79a239f7e898311c5ed6a38c070c641ec56706a295b7e5caf3c55a7296
APACHE_DIST_URLS=https://www.apache.org/dyn/closer.cgi?
action=download&filename=        https://www-us.apache.org/dist/
https://www.apache.org/dist/    https://archive.apache.org/dist/
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
HOME=/root

~$
```

The `kubectl get` command also allows you to retrieve pod metadata using the `-o` switch. The `-o` (or `--output`) switch formats the output of the get command. The output format can be *json*, *yaml*, *wide*, *name*, *template*, *template-file*, *jsonpath*, or *jsonpath-file*. The golang template specification is also used by Docker (more info here: http://golang.org/pkg/text/template/).

For example, to retrieve pod data in YAML, try:

```
~$ kubectl get pod apache -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2024-01-18T01:07:23Z"
  labels:
    run: apache
  name: apache
  namespace: default
  resourceVersion: "1838"
  uid: a13820c3-c6ad-4f57-bc02-ec9133c91a77
spec:
  containers:
  - image: docker.io/httpd:2.2
    imagePullPolicy: IfNotPresent
    name: apache
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-blgw9
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  nodeName: ip-172-31-4-161
  preemptionPolicy: PreemptLowerPriority
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: kube-api-access-blgw9
    projected:
      defaultMode: 420
      sources:
      - serviceAccountToken:
          expirationSeconds: 3607
          path: token
      - configMap:
          items:
          - key: ca.crt
            path: ca.crt
          name: kube-root-ca.crt
      - downwardAPI:
          items:
          - fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
            path: namespace
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2024-01-18T01:07:36Z"
    status: "True"
    type: PodReadyToStartContainers
  - lastProbeTime: null
    lastTransitionTime: "2024-01-18T01:07:23Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2024-01-18T01:07:36Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2024-01-18T01:07:36Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2024-01-18T01:07:23Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID:
containerd://70d5e1cfc69307947f61f75f45832a11c9df5e12bf910d268fa2dca880f0d
8b7
    image: docker.io/library/httpd:2.2
    imageID:
docker.io/library/httpd@sha256:9784d70c8ea466fabd52b0bc8cde84980324f961238
0d22fbad2151df9a430eb
    lastState: {}
    name: apache
    ready: true
    restartCount: 0
    started: true
    state:
      running:
        startedAt: "2024-01-18T01:07:36Z"
  hostIP: 172.31.4.161
  hostIPs:
  - ip: 172.31.4.161
  phase: Running
  podIP: 10.32.0.4
  podIPs:
  - ip: 10.32.0.4
  qosClass: BestEffort
  startTime: "2024-01-18T01:07:23Z"
```

```
~$
```

We can see the entire pod manifest (spec *and* status) using the yaml outputter but like `docker container inspect` this is a lot of information all at once and also like `docker container inspect` it can be parsed for individual properties.

To curl our pod all we need is its IP address. This information is reported dynamically by the kubelet and is part of the pod's "status", so looking under the `status` section of the excerpts above, you will see that "`podIP`" is one of the available status keys.

Using the `jsonpath` outputter or `--template` flag, we can extract just the `podIP`, try it:

```
~$ kubectl get pod apache -o jsonpath='{.status.podIP}' && echo

10.32.0.4

~$
```

Using the above command in a nested shell, store the `podIP` in an environment variable and curl the IP address of the pod to ensure that you can reach the running Apache web server:

```
~$ PIP=$(kubectl get pod apache -o jsonpath='{.status.podIP}') && echo
$PIP

10.32.0.4

~$
```

```
~$ curl -I $PIP

HTTP/1.1 200 OK
Date: Mon, 25 Sep 2023 08:37:46 GMT
Server: Apache/2.2.34 (Unix) mod_ssl/2.2.34 OpenSSL/1.0.1t DAV/2
Last-Modified: Sat, 20 Nov 2004 20:16:24 GMT
ETag: "45118-2c-3e9564c23b600"
Accept-Ranges: bytes
Content-Length: 44
```

```
Content-Type: text/html

~$
```

Your Apache web server is reachable from the pod's IP!

2. Terminating pods

Because pods house running processes on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed. In Kubernetes, users can request deletion and discover when processes terminate.

Go ahead and delete the apache pod:

```
~$ kubectl delete pod apache

pod "apache" deleted

~$ kubectl get pod

No resources found in default namespace.

~$
```

You may have noticed a slight delay between your deletion request and when Kubernetes finished the process. When a user requests deletion of a pod, Kubernetes sends the appropriate termination signal to each container in the pod (either SIGTERM or the container defined stop signal). Kubernetes then waits for a grace period after which, if the pod has not shutdown, the pod is forcefully killed with SIGKILL and the pod is then deleted from the API server. If the Kubelet or the container manager is restarted while waiting for processes to terminate, the termination will be retried with the full grace period.

2.1. CHALLENGE: pod exploration

- Run `kubectl run apache --image docker.io/httpd:latest` to create another `apache` pod
- Use `kubectl describe` to view the new `apache` pod
  - What image is this new apache pod running?
  - Are there any labels?
- Using the `kubectl delete -h` help page, try to delete the new `apache` pod using any labels

3. Pod config files

Let’s try running an nginx container in a pod but this time we’ll create the pod using a YAML configuration file.

```
~$ mkdir -p ~/pods && cd $_

~/pods$
```

Create a spec for a pod named nginxpod that runs the `nginx:1.11` image on port 80. You can easily do so with `kubectl run` with the `--dry-run`, and `-o yaml` flags, sending the output to a file:

```
~/pods$ kubectl run nginxpod --image docker.io/nginx:1.11 --port 80 -o
yaml --dry-run=client > nginxpod.yaml
```

Examine the pod spec:

```
~/pods$ cat nginxpod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginxpod
  name: nginxpod
spec:
  containers:
  - image: nginx:1.11
    name: nginxpod
    ports:
    - containerPort: 80
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

```
~/pods$
```

> N.B. Creating specs imperatively adds the `dnsPolicy`, `restartPolicy`, `status`, `resources`, and `creationTimestamp` keys to the resulting yaml. You can safely remove these keys as Kubernetes will populate default values for them when created. You will also see a label `run: podName` in the metadata.

The key `kind` tells Kubernetes we wish to create a pod. The `metadata` section allows us to define a name for the pod and to apply any other labels we might deem useful.

The `spec` section defines the containers we wish to run in our pod. In our case we will run just a single container based on the nginx image. The `ports` key allows us to share the ports the pod will be using with the orchestration layer.

To have Kubernetes create your new pod you can use the `kubectl create` or `kubectl apply` subcommand. The *create* subcommand will accept a config file via stdin or you can load the config from a file with the `-f` switch (more common). Whereas *apply* loads configs from files or whole directories with `-f`. Try the following:

```
~/pods$ kubectl apply -f nginxpod.yaml

pod/nginxpod created

~/pods$
```

Now list the pods on your cluster:

```
~/pods$ kubectl get pods

NAME       READY   STATUS    RESTARTS   AGE
nginxpod   1/1     Running   0          11s

~/pods$
```

Describe your pod:

```
~/pods$ kubectl describe pod nginxpod

Name:              nginxpod
Namespace:         default
Priority:          0
Service Account:   default
Node:              ip-172-31-4-161/172.31.4.161
Start Time:        Thu, 18 Jan 2024 01:23:01 +0000
Labels:            run=nginxpod
Annotations:       <none>
Status:            Pending
IP:
IPs:               <none>
Containers:
  nginxpod:

...

~/pods$
```

Like we did before with *jsonpath*, extract just the status section's `podIP` value using the `--template` flag and store that in an environment variable:

```
~/pods$ POD=$(kubectl get pod nginxpod --template={{.status.podIP}}) &&
echo $POD

10.32.0.4

~/pods$
```

Like the `jsonpath` outputter, the `--template` flag retrieves to the `podIP` key, which is nested under the `.status` key in the pod spec. The `.key.nestedKey` format is widely used to refer to specific keys in a Kubernetes resource manifest. For example, if you wanted to retrieve the nginxpod's namespace (from the `kubectl get` command above), you would enter the template: `.metadata.namespace`.

Now that we have the pod IP we can try curling our nginx server:

```
~/pods$ curl -I $POD

HTTP/1.1 200 OK
Server: nginx/1.11.13
Date: Thu, 18 Jan 2024 01:24:05 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 04 Apr 2017 15:01:57 GMT
Connection: keep-alive
ETag: "58e3b565-264"
Accept-Ranges: bytes

~/pods$
```

4. Clean up

Now that we have completed our work with the pod we can delete it declaratively using the manifest:

```
~/pods$ kubectl delete -f nginxpod.yaml

pod "nginxpod" deleted

~/pods$
```

```
~/pods$ kubectl get pod

No resources found in default namespace.

~/pods$
```

5. CHALLENGE: a complex pod

Next let’s try creating a pod with a more complex specification.

Create a pod config that describes a pod called `hello` with a:

- container based on an `docker.io/ubuntu:14.04` image
- with an environment variable called `MESSAGE` and a value `hello world`
- Runs the command: `/bin/sh -c "echo $MESSAGE"`
- make sure that the container is *never restarted*

See if you can design this specification on your own.

The pod and container spec documentation can be found here:

- **Pod Spec Reference** - https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/pod-v1/
- **Container Spec Reference** - https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/pod-v1/#Container

`kubectl explain` can help here too! For example, if we want to know what the valid fields for a pod are:

```
~/pods$ kubectl explain pod

KIND:       Pod
VERSION:    v1

DESCRIPTION:
    Pod is a collection of containers that can run on a host. This
resource is
    created by clients and scheduled onto hosts.

FIELDS:
  apiVersion    <string>
    APIVersion defines the versioned schema of this representation of an
object.
    Servers should convert recognized schemas to the latest internal
value, and
    may reject unrecognized values. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#resources

...

~/pods$
```

Since we are interested in the specification or "`pod.spec`", we can use explain to show us the spec fields:

```
~/pods$ kubectl explain pod.spec
```

```
KIND:       Pod
VERSION:    v1

FIELD: spec <PodSpec>

DESCRIPTION:
    Specification of the desired behavior of the pod. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#spec-and-status
    PodSpec is a description of a pod.

FIELDS:
  activeDeadlineSeconds <integer>
    Optional duration in seconds the pod may be active on the node
relative to
    StartTime before the system will actively try to mark it failed and
kill
    associated containers. Value must be a positive integer.

...

~/pods$
```

That's a lot of info! Filter just the objects (without their explanations) with grep:

```
~/pods$ kubectl explain pod.spec | grep "<"

RESOURCE: spec <Object>
  activeDeadlineSeconds <integer>
  affinity  <Object>
  automountServiceAccountToken  <boolean>
  containers    <[]Object> -required-
  dnsConfig <Object>
  dnsPolicy <string>

...

~/pods$
```

`kubectl explain` can navigate through any/all objects/fields, providing explanations, no matter how granular! For example:

```
~/pods$ kubectl explain pod.spec.containers.ports.containerPort

KIND:     Pod
VERSION:  v1

FIELD:    containerPort <integer>
```

```
DESCRIPTION:
     Number of port to expose on the pod's IP address. This must be a
valid port
     number, 0 < x < 65536.

~/pods$
```

When you have the configuration complete, create your pod:

```
~/pods$ kubectl apply -f complex-pod.yaml ### this file only exists after
completing the challenge

pod/hello created

~/pods$
```

List your pods:

```
~/pods$ kubectl get pod

NAME    READY   STATUS      RESTARTS   AGE
hello   0/1     Completed   0          12s

~/pods$
```

We can verify that the container did what it was supposed to do by checking the log output of the pod using the `kubectl logs` subcommand:

```
~/pods$ kubectl logs hello

hello world

~/pods$
```

6. Clean up

Now that we have given our new cluster a good test we can clean up by deleting the pods we have created. The `kubectl delete` subcommand allows you to delete objects you have created in the cluster.

Delete any pods you created:

```
~/pods$ kubectl get pod

NAME    READY   STATUS      RESTARTS   AGE
hello   0/1     Completed   0          60s
```

```
~/pods$
```

```
~/pods$ kubectl delete pod hello

pod "hello" deleted

~/pods$
```

Double-check:

```
~/pods$ kubectl get pod

No resources found in default namespace.

~/pods$
```

You Kubernetes cluster should now be cleaned up:

```
~/pods$ kubectl get services,deployments,replicasets,pods

NAME                  TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes    ClusterIP   10.96.0.1    <none>        443/TCP   39m

~/pods$
```

Be sure to leave the `service/kubernetes` service!

Congratulations, you have completed the lab!