RX-M Cloud Native Consulting

Kubernetes

Lab 3 – Working with Pods

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

In Kubernetes, pods are the smallest deployable units that can be created, scheduled, and managed. A pod corresponds to a collocated group of containers running with a shared context. Within that context, the applications may also have individual cgroup isolations applied. A pod models an application-specific "logical host" in a containerized environment. It may contain one or more applications which are relatively tightly coupled — in a pre-container world, they would have executed on the same physical or virtual host.

The context of the pod can be defined as the conjunction of several Linux namespaces:

- **Network** - applications within the pod have access to the same IP and port space
- **IPC** - applications within the pod can use SystemV IPC or POSIX message queues to communicate
- **UTS** - applications within the pod share a hostname

Applications within a pod can also have access to shared volumes, which are defined at the pod level and made available in each application's file system. Additionally, a pod may define top-level cgroup isolations which form an outer bound to any individual isolation applied to constituent containers.

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 the new node.

As a first step in our exploration we will create a simple single container pod.

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.

```
user@ubuntu:~$ kubectl run apache --generator=run-pod/v1 --image=httpd:2.2

pod/apache created

user@ubuntu:~$
```

Now view the pod:

```
user@ubuntu:~$ kubectl get pod

NAME     READY   STATUS              RESTARTS   AGE
apache   0/1     ContainerCreating   0          5s

user@ubuntu:~$ kubectl get pod

NAME     READY   STATUS    RESTARTS   AGE
apache   1/1     Running   0          14s

user@ubuntu:~$
```

What happened here?

The 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:

```
user@ubuntu:~$ kubectl run -h | grep COMMAND

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

user@ubuntu:~$
```

The *--env* switch sets environment variables (just like the `docker run -e` switch,) the *--port* switch exposes ports for service mapping, the *--replicas* switch sets the number of instances of the pod you would like the cluster to maintain and the *--dry-run* switch (if set to true) allows you to submit the command without executing it to test the parameters.

It doesn't include anything about the `--generator=` flag, let's try again:

```
user@ubuntu:~$ kubectl run -h | grep generator

      --generator='': The name of the API generator to use, see http://kubernetes.io/docs/user-
guide/kubectl-conventions/#generators for a list.
      --service-generator='service/v2': The name of the generator to use for creating a service.
Only used if --expose is true

user@ubuntu:~$
```

Taking a look at the link provided by the help text reveals a number of different generators:

Resource	kubectl command
Pod	kubectl run --generator=run-pod/v1

Many of the generators that used to be valid in previous versions have been deprecated since version 1.12. They will still function if they are invoked, but you will receive a warning each time you do so:

Resource	kubectl command
Replication controller (deprecated)	kubectl run --generator=run/v1
Deployment (deprecated)	kubectl run --generator=extensions/v1beta1
Deployment (deprecated)	kubectl run --generator=apps/v1beta1
Job (deprecated)	kubectl run --generator=job/v1
CronJob (deprecated)	kubectl run --generator=batch/v1beta1
CronJob (deprecated)	kubectl run --generator=batch/v2alpha1

We can use `kubectl run` to create Pods.

For Controllers (RCs, Deployments, Jobs, CronJobs; more on Controllers later) You can also use `kubectl create`, which is what the latest versions are moving towards.

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

```
user@ubuntu:~$ kubectl describe pod apache

Name:         apache
Namespace:    default
Priority:     0
Node:         ubuntu/192.168.228.157
Start Time:   Wed, 08 Jan 2020 13:03:22 -0800
```

```
Labels:       run=apache
Annotations:  <none>
Status:       Running
IP:           10.32.0.4
IPs:
  IP:  10.32.0.4
Containers:
  apache:
    Container ID:   docker://5d79b1d7d769153646d528f06001514340a16b2dcb1bd8a07d6b235b02078fa0
    Image:          httpd:2.2
    Image ID:       docker-
pullable://httpd@sha256:9784d70c8ea466fabd52b0bc8cde84980324f9612380d22fbad2151df9a430eb
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Wed, 08 Jan 2020 13:03:35 -0800
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-7bqf5 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-7bqf5:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-7bqf5
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  2m33s  default-scheduler  Successfully assigned default/apache to ubuntu
  Normal  Pulling    2m32s  kubelet, ubuntu    Pulling image "httpd:2.2"
  Normal  Pulled     2m20s  kubelet, ubuntu    Successfully pulled image "httpd:2.2"
  Normal  Created    2m20s  kubelet, ubuntu    Created container apache
  Normal  Started    2m20s  kubelet, ubuntu    Started container apache

user@ubuntu:~$
```

Read through the *Events* reported for the pod.

You can see that Kubernetes used the Docker Engine 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 node Ubuntu and then the Kubelet on node Ubuntu starts the container.

Use the `docker container ls` subcommand to examine your containers directly on the Docker Engine.

```
user@ubuntu:~$ docker container ls -f "name=apache"

CONTAINER ID        IMAGE                  COMMAND              CREATED             STATUS
PORTS               NAMES
5d79b1d7d769        httpd                  "httpd-foreground"   2 minutes ago       Up 2 minutes
k8s_apache_apache_default_5fd061ed-6706-41ee-b1ba-26181fb78141_0
c3bd17add6fb        k8s.gcr.io/pause:3.1   "/pause"             3 minutes ago       Up 3 minutes
k8s_POD_apache_default_5fd061ed-6706-41ee-b1ba-26181fb78141_0

user@ubuntu:~$
```

Kubernetes incorporates the pod name into the name of each container running in the pod.

Use the `docker container inspect` subcommand to examine the container details for your httpd container:

```
user@ubuntu:~$ docker container inspect $(docker container ls -f "name=apache" --format
'{{.ID}}') | less

[
    {
        "Id": "5d79b1d7d769153646d528f06001514340a16b2dcb1bd8a07d6b235b02078fa0",
        "Created": "2020-01-08T21:03:35.432696743Z",
        "Path": "httpd-foreground",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 89974,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2020-01-08T21:03:35.654326293Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:e06c3dbbfe239c6fca50b6ab6935b3122930fa2eea2136979e5b46ad77ecb685",
        "ResolvConfPath": "/var/lib/docker/containers/c3bd17add6fb526468618ad932
...

q

user@ubuntu:~$
```

Notice that Kubernetes injects a standard set of environment variables into the containers of a pod providing the location of the cluster API service. If you have not done so already, use `apt` to install the `jq` JSON processor:

```
user@ubuntu:~$ sudo apt-get install jq -y

...

user@ubuntu:~$ docker container inspect $(docker container ls -f "name=apache" --format
'{{.ID}}') \
| jq -r '.[0].Config.Env[]'

KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
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
PATH=/usr/local/apache2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HTTPD_PREFIX=/usr/local/apache2
HTTPD_VERSION=2.2.34
HTTPD_SHA256=e53183d5dfac5740d768b4c9bea193b1099f4b06b57e5f28d7caaf9ea7498160
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/

user@ubuntu:~$
```

Kubernetes has also added several labels to the container, for example you can see that the pod is running in the default namespace.

```
user@ubuntu:~$ docker container inspect $(docker container ls -f "name=apache" --format
'{{.ID}}') \
```

```
| jq -r '.[0].Config.Labels'

{
  "annotation.io.kubernetes.container.hash": "c3e9fd0c",
  "annotation.io.kubernetes.container.restartCount": "0",
  "annotation.io.kubernetes.container.terminationMessagePath": "/dev/termination-log",
  "annotation.io.kubernetes.container.terminationMessagePolicy": "File",
  "annotation.io.kubernetes.pod.terminationGracePeriod": "30",
  "io.kubernetes.container.logpath": "/var/log/pods/default_apache_5fd061ed-6706-41ee-b1ba-
26181fb78141/apache/0.log",
  "io.kubernetes.container.name": "apache",
  "io.kubernetes.docker.type": "container",
  "io.kubernetes.pod.name": "apache",
  "io.kubernetes.pod.namespace": "default",
  "io.kubernetes.pod.uid": "5fd061ed-6706-41ee-b1ba-26181fb78141",
  "io.kubernetes.sandbox.id": "c3bd17add6fb526468618ad932f15a306bf946f44d6573d4eb72c5b96bf9fbde"
}

user@ubuntu:~$
```

`curl` the IP address of the web server to ensure that you can reach the running Apache web server. Don't forget -complete.

```
user@ubuntu:~$ PIP=$(kubectl get pod -o jsonpath='{.items[*].status.podIP}') && echo $PIP

10.32.0.4

user@ubuntu:~$
```

```
user@ubuntu:~$ curl -I $PIP

HTTP/1.1 200 OK
Date: Wed, 08 Jan 2020 21:09:43 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: "43bc-2c-3e9564c23b600"
Accept-Ranges: bytes
Content-Length: 44
Content-Type: text/html

user@ubuntu:~$
```

- How can you discover the address of the Apache web server running inside the container?
  - try
    `docker container inspect $(docker container ls -f "ancestor=httpd:2.2" --format '{{.ID}}') |grep IPAddress`
- Why can't you find the IP address in the container inspect data?
- What is the value of the Apache container's HostConfig.NetworkMode setting in the inspect data?
  - hint:
    `docker container inspect $(docker container ls -f "ancestor=httpd:2.2" --format '{{.ID}}') -f '{{.HostConfig.NetworkMode}}'`
- What container does this reference?
  - `docker container ls -f "ID=<insert_ID_returned_by_the_previous_command_here>"`
- What other keys in the apache container inspect has the same value and why?

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. 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.

Go ahead and delete the apache pod:

```
user@ubuntu:~/pods$ kubectl delete pod apache

pod "apache" deleted

user@ubuntu:~/pods$ kubectl get deployment,replicaset,pods

No resources found in default namespace.

user@ubuntu:~/pods$
```

While `kubectl run` is handy for quickly starting a single image based pod it is not the most flexible or repeatable way to create pods. Next we'll take a look at a more useful way of starting pods, from a configuration file.

2. Pod Config Files

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 friendly system. They can also save a lot of typing if you would like to deploy complex pods or entire applications.

Let's try running an nginx container in a pod but this time we'll create the pod using a YAML configuration file. Create the following config file with your favorite editor (as long as it is nano) in a new "pods" working directory.

```
user@ubuntu:~$ mkdir pods && cd pods

user@ubuntu:~/pods$
```

```
user@ubuntu:~/pods$ nano nginxpod.yaml && cat nginxpod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: nginxpod
spec:
  containers:
  - name: nginx
    image: nginx:1.11
    ports:
    - containerPort: 80

user@ubuntu:~/pods$
```

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. More on this later.

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 with `-f` or whole directories with `-k`. Try the following:

```
user@ubuntu:~/pods$ kubectl apply -f nginxpod.yaml

pod/nginxpod created

user@ubuntu:~/pods$
```

Now list the pods on your cluster:

```
user@ubuntu:~/pods$ kubectl get pods

NAME       READY   STATUS    RESTARTS   AGE
nginxpod   1/1     Running   0          5s
```

```
user@ubuntu:~/pods$
```

Describe your pod:

```
user@ubuntu:~/pods$ kubectl describe pod nginxpod

Name:         nginxpod
Namespace:    default
Priority:     0
Node:         ubuntu/192.168.228.157
Start Time:   Wed, 08 Jan 2020 13:11:32 -0800
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":
{},"name":"nginxpod","namespace":"default"},"spec":{"containers":[{"image":"ngin...
Status:       Running
IP:           10.32.0.4
IPs:
  IP:  10.32.0.4
Containers:
  nginx:
    Container ID:   docker://5d6d28322f832ff9e1b9be308e2f973ea8394d66f3caf0168bd8f4da330e0fee
    Image:          nginx:1.11
    Image ID:       docker-
pullable://nginx@sha256:e6693c20186f837fc393390135d8a598a96a833917917789d63766cab6c59582
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Wed, 08 Jan 2020 13:11:34 -0800
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-7bqf5 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-7bqf5:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-7bqf5
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  14s   default-scheduler  Successfully assigned default/nginxpod to ubuntu
  Normal  Pulled     13s   kubelet, ubuntu    Container image "nginx:1.11" already present on
machine
  Normal  Created    13s   kubelet, ubuntu    Created container nginx
  Normal  Started    12s   kubelet, ubuntu    Started container nginx

user@ubuntu:~/pods$
```

The `kubectl get` command 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:

```
user@ubuntu:~/pods$ kubectl get pod nginxpod -o yaml | head
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":
{},"name":"nginxpod","namespace":"default"},"spec":{"containers":
[{"image":"nginx:1.11","name":"nginx","ports":[{"containerPort":80}]}]}}
  creationTimestamp: "2020-01-08T21:11:32Z"
  name: nginxpod
  namespace: default
  resourceVersion: "4595"

user@ubuntu:~/pods$
```

You can use a template to extract just the data you want.

For example, to extract just the status section's podIP value try:

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

10.32.0.4

user@ubuntu:~/pods$
```

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

```
user@ubuntu:~/pods$ curl -I $POD

HTTP/1.1 200 OK
Server: nginx/1.11.13
Date: Wed, 08 Jan 2020 21:12:43 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

user@ubuntu:~/pods$
```

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

```
user@ubuntu:~/pods$ kubectl delete pod nginxpod

pod "nginxpod" deleted

user@ubuntu:~/pods$
```

```
user@ubuntu:~/pods$ kubectl get deployment,replicaset,pods

No resources found in default namespace.

user@ubuntu:~/pods$
```

3. A Complex Pod

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

Create a pod config that describes a pod with a:

- container based on an *ubuntu:14.04* image,
- with an environment variable called "MESSAGE" and a

- command that will `echo` that message to stdout
- 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/generated/kubernetes-api/v1.16/#pod-v1-core
- **Container Spec Reference** - https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/#container-v1-core

Create your pod when you have the configuration complete:

```
user@ubuntu:~/pods$ kubectl apply -f cpod.yaml

pod/hello created

user@ubuntu:~/pods$
```

List your pods:

```
user@ubuntu:~/pods$ kubectl get pod

NAME      READY     STATUS      RESTARTS   AGE
hello     0/1       Completed   0          28s

user@ubuntu:~/pods$
```

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

```
user@ubuntu:~/pods$ kubectl logs hello

hello world

user@ubuntu:~/pods$
```

- Remove the hello pod

4. Pods and Linux Namespaces

Using the pod spec reference as a guide again, modify your nginx config (nginxpod.yaml) from step 2 so that it runs all of the containers in the pod in the **host network namespace** (hostNetwork). Create a new pod from your updated config.

First copy the old pod spec:

```
user@ubuntu:~/pods$ cp nginxpod.yaml nginxpod-namespace.yaml

user@ubuntu:~/pods$
```

Next modify the copy to use the hostNetwork (use the spec reference if you need help thinking through the edits you will need to make):

```
user@ubuntu:~/pods$ nano nginxpod-namespace.yaml

...

user@ubuntu:~/pods$
```

Run the new pod:

```
user@ubuntu:~/pods$ kubectl apply -f nginxpod-namespace.yaml

pod/nginxpod created

user@ubuntu:~/pods$
```

Now display your pod status in yaml output:

```
user@ubuntu:~/pods$ kubectl get pod nginxpod -o yaml

apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":
{},"name":"nginxpod","namespace":"default"},"spec":{"containers":
[{"image":"nginx:1.11","name":"nginx","ports":[{"containerPort":80}]}],"hostNetwork":true}}
  creationTimestamp: "2020-01-08T21:16:14Z"
  name: nginxpod
  namespace: default
  resourceVersion: "4980"
  selfLink: /api/v1/namespaces/default/pods/nginxpod
  uid: 07c33157-51d6-4783-a5a6-2e9a06adebfc
spec:
  containers:
  - image: nginx:1.11
    imagePullPolicy: IfNotPresent
    name: nginx
    ports:
    - containerPort: 80
      hostPort: 80
      protocol: TCP
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-7bqf5
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  hostNetwork: true
  nodeName: ubuntu
  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: default-token-7bqf5
    secret:
      defaultMode: 420
      secretName: default-token-7bqf5
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2020-01-08T21:16:14Z"
```

```
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2020-01-08T21:16:15Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2020-01-08T21:16:15Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2020-01-08T21:16:14Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: docker://efd6ef408bc2bb6fa6c4703612ac7716cf1fc899751a5509027c3d07b42e2833
    image: nginx:1.11
    imageID: docker-
pullable://nginx@sha256:e6693c20186f837fc393390135d8a598a96a833917917789d63766cab6c59582
    lastState: {}
    name: nginx
    ready: true
    restartCount: 0
    started: true
    state:
      running:
        startedAt: "2020-01-08T21:16:14Z"
  hostIP: 192.168.228.157
  phase: Running
  podIP: 192.168.228.157
  podIPs:
  - ip: 192.168.228.157
  qosClass: BestEffort
  startTime: "2020-01-08T21:16:14Z"

user@ubuntu:~/pods$
```

If your pod is running in the host network namespace you should see that the pod (*podIP*) and host IP (*hostIP*) address are identical.

```
user@ubuntu:~/pods$ kubectl get pod nginxpod -o yaml | grep -E "(host|pod)IP"

  hostIP: 192.168.228.157
  podIP: 192.168.228.157
  podIPs:

user@ubuntu:~/pods$
```

Namespace select-ability allows you have the benefits of container deployment while still empowering infrastructure tools to see and manipulate host based networking features.

Try curling your pod using the host IP address:

```
user@ubuntu:~/pods$ curl -I 192.168.228.157

HTTP/1.1 200 OK
Server: nginx/1.11.13
Date: Wed, 08 Jan 2020 21:17:26 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

user@ubuntu:~/pods$
```

- Clean up the resources (pods, etc.)

5. Multi-container pod

In this step we'll experiment with multi-container pods. Keep in mind that by default all containers in a pod share the same network, uts, and ipc namespace.

You can use the *--validate* switch with the `kubectl apply` subcommand to verify your pod config, however the *apply* command will try to create the config regardless (work to be done here).

Create a new Pod config which:

- runs two ubuntu:14.04 containers, named hello1 and hello2
- both executing the command line "tail -f /dev/null"
- and then apply it with the validate switch

You can start from an existing pod config if you like:

```
user@ubuntu:~/pods$ cp cpod.yaml two.yaml

user@ubuntu:~/pods$
```

Then edit the new config to meet the requirements:

```
user@ubuntu:~/pods$ nano two.yaml

...

user@ubuntu:~/pods$
```

Finally, create the pod:

```
user@ubuntu:~/pods$ kubectl apply -f two.yaml --validate

pod/two created

user@ubuntu:~/pods$
```

If you got it right the first time the pod will simply run. If not you will get a descriptive error.

Issue the `kubectl get pods` command.

- How many containers are running in your new pod?
- How can you tell?

Use the `kubectl describe pod` subcommand on your new pod.

- What is the ip address of the first container?
- What is the ip address of the second container?

Shell into the first container to explore its context.

e.g. `kubectl exec -c hello1 -it two /bin/bash`

Run the `ps -ef` command inside the container.

- What processes are running in the container?
- What is the container's host name?

e.g.

```
user@ubuntu:~/pods$ kubectl exec -c hello1 -it two /bin/bash

root@two:/# ps -ef

UID         PID   PPID  C STIME TTY          TIME CMD
```

```
root           1      0  0 21:20 ?        00:00:00 sh -c tail -f /dev/null
root          11      1  0 21:20 ?        00:00:00 tail -f /dev/null
root          12      0  0 21:20 pts/0    00:00:00 /bin/bash
root          26     12  0 21:21 pts/0    00:00:00 ps -ef

root@two:/#
```

Run the `ip a` command inside the container.

- What is the MAC address of eth0?
- What is the IP address

Create a file in the root directory and exit the container:

```
root@two:/# echo "Hello" > TEST

root@two:/# exit
```

Kubernetes executed our last command in the first container in the pod. We now want to open a shell into the second container. To do this we can use the *-c* switch. Exec a shell into the second container using the *-c* switch and the name of the second container:

e.g. `kubectl exec -c hello2 -it two /bin/bash`

- Is the TEST file you created previously there?
- What is the host name in this container?
- What is the MAC address of eth0?
- What is the IP address?
- Which of the following namespaces are shared across the containers in the pod?
  - User
  - Process
  - UTS (hostname)
  - Network
  - IPC
  - Mount (filesystem)

Clean up the resources (pods, etc.)

6. Resource Requirements

Next let's explore resource requirements. Kubernetes configs allow you to specify requested levels of memory and cpu. You can also assign limits. Requests are used when scheduling the pod to ensure that it is place on a host with enough resources free. Limits are configured in the kernel cgroups to constrain the runtime use of resources by the pod.

Create a new pod config (*limit.yaml*) like the following:

```
user@ubuntu:~/pods$ nano limit.yaml && cat limit.yaml

apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: ".25"
      limits:
        memory: "128Mi"
        cpu: ".5"
  - name: wp
    image: wordpress
    resources:
```

```
      requests:
        memory: "64Mi"
        cpu: ".25"
      limits:
        memory: "128Mi"
        cpu: ".5"

user@ubuntu:~/pods$
```

This config will run a pod with the Wordpress image and the MySql image with explicit requested resource levels and explicit resource constraints.

Before you create the pod verify the resources on your node:

```
user@ubuntu:~/pods$ kubectl describe $(kubectl get node -o name) | grep -A 6 Allocated

Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests    Limits
  --------           --------    ------
  cpu                770m (38%)  0 (0%)
  memory             140Mi (3%)  340Mi (8%)
  ephemeral-storage  0 (0%)      0 (0%)

user@ubuntu:~/pods$
```

Examine the Capacity field for your node.

```
user@ubuntu:~/pods$ kubectl get $(kubectl get node -o name) -o json | jq .status.capacity

{
  "cpu": "2",
  "ephemeral-storage": "18447100Ki",
  "hugepages-1Gi": "0",
  "hugepages-2Mi": "0",
  "memory": "4028884Ki",
  "pods": "110"
}

user@ubuntu:~/pods$
```

If you are unfamiliar with `jq`, here is an example to list keys nested in the metadata object.

```
user@ubuntu:~/pods$ kubectl get $(kubectl get nodes -o name | head -1) -o json | jq '.metadata |
keys'

[
  "annotations",
  "creationTimestamp",
  "labels",
  "name",
  "resourceVersion",
  "selfLink",
  "uid"
]

user@ubuntu:~/pods$
```

Next examine the Allocated resources section.

- Will the node be able accept the scheduled pod?

Now create the new pod and verify its construction:

```
user@ubuntu:~$ kubectl apply -f limit.yaml
```

```
pod/frontend created

user@ubuntu:~/pods$ kubectl get pods

NAME           READY     STATUS              RESTARTS   AGE
frontend       0/2       ContainerCreating   0          5s
user@ubuntu:~/pods$
```

Use the `kubectl describe pod frontend` or `kubectl get events | grep -i frontend` command to display the events for your new pod. You may see the image pulling. This means the Docker daemon is pulling the image in the background. You can monitor the pull by issuing the `docker pull` subcommand for the same image on the pulling host:

```
user@ubuntu:~/pods$ docker image pull wordpress

Using default tag: latest
latest: Pulling from library/wordpress
Digest: sha256:73e8d8adf491c7a358ff94c74c8ebe35cb5f8857e249eb8ce6062b8576a01465
Status: Image is up to date for wordpress:latest

user@ubuntu:~/pods$
```

Once the image has pulled, check the pod status:

```
user@ubuntu:~/pods$ kubectl get pods

NAME       READY     STATUS    RESTARTS   AGE
frontend   1/2       Error     2          74s

user@ubuntu:~/pods$
```

It errored out (or may be in a CrashLoopBack cycle)! Try to diagnose any issues using `kubectl logs frontend`. (Hint: Some containerized applications need more configuration than others).

Regardless of whether the application is actually running, the pod still has resources set aside for it on the host.

Rerun the `kubectl describe $(kubectl get node -o name)` command to redisplay your node resource usage:

```
user@ubuntu:~/pods$ kubectl describe $(kubectl get node -o name)

...

user@ubuntu:~/pods$ kubectl describe $(kubectl get node -o name) | grep -A 6 Allocated

Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests     Limits
  --------           --------     ------
  cpu                1270m (63%)  1 (50%)
  memory             268Mi (6%)   596Mi (15%)
  ephemeral-storage  0 (0%)       0 (0%)

user@ubuntu:~/pods$
```

Can you see the impact of the new pod on the host? How does it correspond with the resource limits and requests you listed in the pod spec?

Clean Up

Delete the frontend pod after examining its resource usage.

```
user@ubuntu:~/pods$ kubectl delete pod frontend
```

```
pod "frontend" deleted

user@ubuntu:~/pods$
```

Congratulations, you have completed the lab!

Selected Answers

Pod cpod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello
spec:  # specification of the pod's contents
  restartPolicy: Never
  containers:
  - name: hellocont
    image: "ubuntu:14.04"
    env:
    - name: MESSAGE
      value: "hello world"
    command: ["/bin/sh","-c"]
    args: ["/bin/echo \"${MESSAGE}\""]
```

Pod nginxpod-namespace.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginxpod
spec:
  hostNetwork: true
  containers:
  - name: nginx
    image: nginx:1.11
    ports:
    - containerPort: 80
```

Pod two.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: two
spec:
  restartPolicy: Never
  containers:
  - name: hello1
    image: "ubuntu:14.04"
    command: ["/usr/bin/tail"]
    args: ["-f", "/dev/null"]
  - name: hello2
    image: "ubuntu:14.04"
    command: ["/usr/bin/tail"]
    args: ["-f", "/dev/null"]
```