Cloud Foundry Container Runtime

Lab 11 - 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:

- **PID** - applications within the pod can see each other's processes (as of Docker 1.12)
- **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 s 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 --image=httpd:2.2

deployment.apps "apache" created
user@ubuntu:~$
```

Now view the Deployment:

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

NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
apache    1         1         1            1           17s
user@ubuntu:~$
```

Great, our pod is deployed. Now view the replica set:

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

NAME               DESIRED   CURRENT   READY     AGE
apache-bd64d46f6   1         1         1         17s
user@ubuntu:~$
```

and pods:

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

NAME                     READY     STATUS    RESTARTS   AGE
apache-bd64d46f6-f8kj4   1/1       Running   0          27s
user@ubuntu:~$
```

What happened here?

The run command takes a name and an image as parameters and then rather than generate a pod it generates a Deployment which defines a replica set with a pod template including the image you specified.

The replica set begins with a scale of one, causing an instance of the pod template to get scheduled. While this is usually what you want, it could be counterproductive if you wanted to run a batch style pod that is expected to exit. Replica sets ensure that some number of instances of the pod template are always running.

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.

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

```
user@ubuntu:~$ kubectl describe pod apache-bd64d46f6-f8kj4

Name:           apache-bd64d46f6-f8kj4
Namespace:      default
Node:           ubuntu/172.16.151.229
Start Time:     Wed, 28 Mar 2018 20:47:18 -0700
Labels:         pod-template-hash=682080292
                run=apache
Annotations:    kubernetes.io/created-by={"kind":"SerializedReference","apiVersion":"v1","reference":
{"kind":"ReplicaSet","namespace":"default","name":"apache-bd64d46f6","uid":"b6de92cb-d93f-11e7-a277-
000c29ae8ddc","...
Status:         Running
IP:             10.32.0.4
Created By:     ReplicaSet/apache-bd64d46f6
Controlled By:  ReplicaSet/apache-bd64d46f6
Containers:
  apache:
    Container ID:   docker://c708f5af131fc4d3fe9722b3fe95c84d0d90dd440ac3b65272b87ecb5ff10675
    Image:          httpd:2.2
    Image ID:       docker-pullable://httpd@sha256:4eceb4a355a19d6e33b51fbabe5f1a236bd7c6e370047877214459b8cf3c2362
    Port:           <none>
    State:          Running
      Started:      Wed, 28 Mar 2018 20:47:18 -0700
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-2kmhb (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Volumes:
  default-token-2kmhb:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-2kmhb
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.alpha.kubernetes.io/notReady:NoExecute for 300s
                 node.alpha.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason                 Age   From               Message
  ----    ------                 ----  ----               -------
  Normal  Scheduled              1m    default-scheduler  Successfully assigned apache-bd64d46f6-f8kj4 to ubuntu
  Normal  SuccessfulMountVolume  1m    kubelet, ubuntu    MountVolume.SetUp succeeded for volume "default-token-
2kmhb"
  Normal  Pulling                1m    kubelet, ubuntu    pulling image "httpd:2.2"
  Normal  Pulled                 1m    kubelet, ubuntu    Successfully pulled image "httpd:2.2"
  Normal  Created                1m    kubelet, ubuntu    Created container
  Normal  Started                1m    kubelet, ubuntu    Started container
user@ubuntu:~$
```

Read through the *Events* reported for the pod.

You can also 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
c708f5af131f        httpd                                     "httpd-foreground"   About a minute ago   Up About a
minute                      k8s_apache_apache-bd64d46f6-f8kj4_default_b6e04bbe-d93f-11e7-a277-000c29ae8ddc_0
6ee5c319a3ac        gcr.io/google_containers/pause-amd64:3.0   "/pause"            About a minute ago   Up About a
minute                      k8s_POD_apache-bd64d46f6-f8kj4_default_b6e04bbe-d93f-11e7-a277-000c29ae8ddc_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 $(docker container ls -f "name=apache" --format '{{.ID}}') | less
[
    {
        "Id": "c708f5af131fc4d3fe9722b3fe95c84d0d90dd440ac3b65272b87ecb5ff10675",
        "Created": "2017-12-04T22:09:06.289249262Z",
        "Path": "httpd-foreground",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 96991,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2017-12-04T22:09:06.489950886Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:5a312d4f55c5159f67add782089d42e37bffabdedd0d5ae6ee34ae56cadf495e",
        "ResolvConfPath":
"/var/lib/docker/containers/6ee5c319a3ac49b5e062bdfa24c3bc2103b1ae1000cf5fc0161daaa0c7a3a47a/resolv.conf",
        "HostnamePath":
"/var/lib/docker/containers/6ee5c319a3ac49b5e062bdfa24c3bc2103b1ae1000cf5fc0161daaa0c7a3a47a/hostname",
        "HostsPath": "/var/lib/kubelet/pods/b6e04bbe-d93f-11e7-a277-000c29ae8ddc/etc-hosts",
        "LogPath":
"/var/lib/docker/containers/c708f5af131fc4d3fe9722b3fe95c84d0d90dd440ac3b65272b87ecb5ff10675/c708f5af131fc4d3fe9722b
3fe95c84d0d90dd440ac3b65272b87ecb5ff10675-json.log",
        "Name": "/k8s_apache_apache-bd64d46f6-f8kj4_default_b6e04bbe-d93f-11e7-a277-000c29ae8ddc_0",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
        "MountLabel": "",
        "ProcessLabel": "",
        "AppArmorProfile": "docker-default",
        "ExecIDs": null,
...
user@ubuntu:~$
```

Notice that Kubernetes injects a standard set of environment variables into the containers of a pod providing the location of the cluster API service.

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

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
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
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
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": "85b9d29",
  "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/b6e04bbe-d93f-11e7-a277-000c29ae8ddc/apache_0.log",
  "io.kubernetes.container.name": "apache",
  "io.kubernetes.docker.type": "container",
  "io.kubernetes.pod.name": "apache-bd64d46f6-f8kj4",
  "io.kubernetes.pod.namespace": "default",
  "io.kubernetes.pod.uid": "b6e04bbe-d93f-11e7-a277-000c29ae8ddc",
  "io.kubernetes.sandbox.id": "6ee5c319a3ac49b5e062bdfa24c3bc2103b1ae1000cf5fc0161daaa0c7a3a47a"
}
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:~$ kubectl describe pod apache-bd64d46f6-f8kj4 | grep IP

IP:             10.32.0.4
user@ubuntu:~$
```

```
user@ubuntu:~$ curl -I 10.32.0.4

HTTP/1.1 200 OK
Date: Mon, 04 Dec 2017 22:12:42 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: "43e1-2c-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?
- 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 "name=apache" --format '{{.ID}}') -f '{{.HostConfig.NetworkMode}}'`
- What container does this reference?
- 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.

Try deleting your pod (use the pod name displayed by `kubectl get pods`).

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

NAME                     READY     STATUS    RESTARTS   AGE
apache-bd64d46f6-f8kj4   1/1       Running   0          4m
user@ubuntu:~$
```

```
user@ubuntu:~$ kubectl delete pod apache-bd64d46f6-f8kj4

pod "apache-bd64d46f6-f8kj4" deleted
user@ubuntu:~$
```

Now display the running pods.

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

NAME                     READY     STATUS    RESTARTS   AGE
apache-bd64d46f6-s7brc   1/1       Running   0          6s
user@ubuntu:~$
```

What is happening?

Our pod was not created directly, it was created by a replica set with the replication factor set to 1. An important thing to remember: `kubectl run` creates a deployment with a replica set which then creates a pod of scale 1. When you terminate the pod you violate the desired state. The scale status for the pod becomes 0 but the target is 1. So the replica set schedules a replacement.

You might ask: "why does kubernetes let me delete the pod if it will just restart it?". There are many reasons you might want to delete a given pod. Perhaps it has problems and you want to generate a new replacement. Perhaps the current node has problems and you want Kubernetes to reschedule this particular pod somewhere else.

If you want to run a batch job, just once, `kubectl run` is not the right command. Kubernetes 1.2.2 offers a *"job"* controller which is perfectly suited for run-once style pods (more on this later.)

In the above case, when the rs saw no pods with the label "run=apache" running, it started a new one. To actually terminate our pod permanently we must delete the deployment, the deployment controls the replica set, the replica set controls the pods.

Try it:

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

NAME            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/apache   1         1         1            1           4m

NAME                   DESIRED   CURRENT   READY     AGE
rs/apache-bd64d46f6    1         1         1         4m
user@ubuntu:~$
```

```
user@ubuntu:~$ kubectl delete deployment apache

deployment.extensions "apache" deleted
user@ubuntu:~$
```

Now list the running deployments, rs, and pods:

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

No resources found.
user@ubuntu:~$
```

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 vim) in a new "pods" working directory.

```
user@ubuntu:~$ mkdir pods

user@ubuntu:~$
```

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

user@ubuntu:~/pods$
```

```
user@ubuntu:~/pods$ vi nginxpod.yaml
user@ubuntu:~/pods$ 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. This is in contrast to the *run* subcommand which created a deployment and replica set that in turn created 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` 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). Try the following:

```
user@ubuntu:~/pods$ kubectl create -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          15s

user@ubuntu:~/pods$
```

Great, our pod is up and running. Now check for deployments and replica sets:

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

No resources found.
user@ubuntu:~/pods$
```

Describe your pod:

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

Name:         nginxpod
Namespace:    default
Node:         ubuntu/172.16.151.229
Start Time:   Wed, 28 Mar 2018 21:20:45 -0700
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           10.32.0.4
Containers:
  nginx:
    Container ID:   docker://1d1ebe283310f53c921a51d323973ae148bc8cb41a78e59306d0399c31437383
    Image:          nginx:1.11
    Image ID:       docker-pullable://nginx@sha256:e6693c20186f837fc393390135d8a598a96a833917917789d63766cab6c59582
    Port:           80/TCP
    State:          Running
      Started:      Mon, 04 Dec 2017 14:14:30 -0800
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-2kmhb (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Volumes:
  default-token-2kmhb:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-2kmhb
```

```
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.alpha.kubernetes.io/notReady:NoExecute for 300s
                 node.alpha.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason                 Age   From               Message
  ----    ------                 ----  ----               -------
  Normal  Scheduled              24s   default-scheduler  Successfully assigned nginxpod to ubuntu
  Normal  SuccessfulMountVolume  24s   kubelet, ubuntu    MountVolume.SetUp succeeded for volume "default-token-
2kmhb"
  Normal  Pulled                 23s   kubelet, ubuntu    Container image "nginx:1.11" already present on machine
  Normal  Created                23s   kubelet, ubuntu    Created container
  Normal  Started                23s   kubelet, ubuntu    Started container
user@ubuntu:~/pods$
```

In previous versions of kubernetes you would see `Controllers: <none>`, but in our version it is removed/missing?!

The `kubectl` 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:
  creationTimestamp: 2018-03-29T04:20:45Z
  name: nginxpod
  namespace: default
  resourceVersion: "12743"
  selfLink: /api/v1/namespaces/default/pods/nginxpod
  uid: 7f4a2bbf-d940-11e7-a277-000c29ae8ddc
spec:
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$ kubectl get pod nginxpod -o template --template={{.status.podIP}} && echo

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 10.32.0.4

HTTP/1.1 200 OK
Server: nginx/1.11.13
Date: Mon, 16 Oct 2017 01:19:46 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.
user@ubuntu:~/pods$
```

Because we did not create a replica set there is no nanny to restart our pod when we delete it.

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/api-reference/v1.8/#podspec-v1-core
- **Container Spec Reference** - https://kubernetes.io/docs/api-reference/v1.8/#container-v1-core

Create your pod when you have the configuration complete:

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

pod "hello" created

user@ubuntu:~/pods$
```

One possible solution might look like this:

```
user@ubuntu:~/pods$ cat 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 \"${MESSAGE}\""]
user@ubuntu:~/pods$
```

List your pods:

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

NAME      READY     STATUS      RESTARTS   AGE
hello     0/1       Completed   0          28s
user@ubuntu:~/pods$
```

In older versions of Kubernetes, when a container completed it would not show up:

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

No resources found, use --show-all to see completed objects.
user@ubuntu:~/pods$
```

Using the `--show-all` switch was necessary to see the exited pods:

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

NAME      READY     STATUS      RESTARTS   AGE
hello     0/1       Completed   0          43s
user@ubuntu:~/pods$
```

But in newer versions, it is deprecated:

```
user@ubuntu:~$ kubectl get pod ---show-all
```

```
Flag --show-all has been deprecated, will be removed in an upcoming release
NAME      READY     STATUS      RESTARTS   AGE
hello     0/1       Completed   0          31s
user@ubuntu:~$
```

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 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.v2.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$ vim nginxpod.v2.yaml

...

user@ubuntu:~/pods$
```

Run the new pod:

```
user@ubuntu:~/pods$ kubectl create -f nginxpod.v2.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:
  creationTimestamp: 2018-03-29T04:20:45Z
  name: nginxpod
  namespace: default
  resourceVersion: "13010"
  selfLink: /api/v1/namespaces/default/pods/nginxpod
  uid: 0308a21b-d941-11e7-a277-000c29ae8ddc
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-2kmhb
      readOnly: true
  dnsPolicy: ClusterFirst
```

```
  hostNetwork: true
  nodeName: ubuntu
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.alpha.kubernetes.io/notReady
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.alpha.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: default-token-2kmhb
    secret:
      defaultMode: 420
      secretName: default-token-2kmhb
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: 2018-03-29T04:20:49Z
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: 2018-03-29T04:20:50Z
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: 2018-03-29T04:20:59
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: docker://d980e9a2dc1ebe33dba6e1af8e5e3c90d0001bef93d7ee1bf458a757c37aeae4
    image: nginx:1.11
    imageID: docker-pullable://nginx@sha256:e6693c20186f837fc393390135d8a598a96a833917917789d63766cab6c59582
    lastState: {}
    name: nginx
    ready: true
    restartCount: 0
    state:
      running:
        startedAt: 2018-03-29T04:20:45Z
  hostIP: 172.16.151.229
  phase: Running
  podIP: 172.16.151.229
  qosClass: BestEffort
  startTime: 2018-03-29T04:20:45Z
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: 172.16.151.229
  podIP: 172.16.151.229
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 172.16.151.229

HTTP/1.1 200 OK
Server: nginx/1.11.13
Date: Mon, 24 Mar 2018 22:19:04 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 create` subcommand to verify your pod config, however the *create* 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,
- both executing the command line "tail -f /dev/null"
- and then create 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$ vim two.yaml

...

user@ubuntu:~/pods$
```

Finally, create the pod:

```
user@ubuntu:~/pods$ kubectl create -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 22:21 ?        00:00:00 tail -f /dev/null
root         22      0  0 22:23 pts/0    00:00:00 /bin/bash
root         36     22  0 22:23 pts/0    00:00:00 ps -ef
root@two:/# exit
exit
user@ubuntu:~/pods$
```

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:

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

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 nodes/ubuntu | grep -A 4 Allocated

Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  ------------  ----------  ---------------  -------------
  830m (41%)    0 (0%)      110Mi (2%)       170Mi (4%)
```

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

Examine the Capacity field for your node.

```
user@ubuntu:~/pods$ kubectl get nodes/ubuntu -o json | jq .status.capacity

{
  "cpu": "2",
  "ephemeral-storage": "18447100Ki",
  "hugepages-1Gi": "0",
  "hugepages-2Mi": "0",
  "memory": "2029876Ki",
  "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 create -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: sha256:8fd3cad0d1a9291db828ea74a7aee4cc01ff94b5ee17df493279e4d673cab56f
Status: Image is up to date for wordpress:latest
user@ubuntu:~/pods$
```

Once the image has pulled you may see problems (Error or CrashLoopBackOff).

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

NAME       READY     STATUS    RESTARTS   AGE
frontend   1/2       Error     2          1m
user@ubuntu:~/pods$
```

Try to diagnose and repair any issues, hint use `kubectl logs frontend -c <container name>`.

Rerun the `kubectl describe node/ubuntu` command to redisplay your node resource usage.

```
user@ubuntu:~/pods$ kubectl describe nodes/ubuntu
```

```
...
user@ubuntu:~/pods$ kubectl describe nodes/ubuntu | grep -A 4 Allocated

Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  ------------  ----------  ---------------  -------------
  1330m (66%)   1 (50%)     238Mi (6%)       426Mi (11%)
user@ubuntu:~/pods$
```

Can you see the impact of the new pod? Delete the pod after examining its resource usage.

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

pod "frontend" deleted
user@ubuntu:~/pods$
```

Congratulations, you have completed the Kubernetes working with pods lab!