KUBERNETES PODS: AN INTRODUCTION



In this article, I'm going to explain Kubernetes pods, use cases, and lifecycle, as well as how to use pods to deploy an application.

This post assumes you understand the purpose of Kubernetes and you have minikube and kubectl installed.

(This article is part of our Kubernetes Guide. Use the right-hand menu to navigate.)

What is a K8s pod?

Of object models in Kubernetes, the pod is the smallest building block. Within a cluster, a pod represents a process that's running. The inside of a pod can have one or more containers. Those within a single pod share:

- A unique network IP
- Network
- Storage
- Any additional specifications you've applied to the pod

Another way to think of a pod—a "logical host" that is specific to your application and holds one or more tightly-coupled containers. For example, say we have an **app-container** and a **logging-container** in a pod. The only job of the logging-container is to pull logs from the app-container. Locating your containers in a pod eliminates extra communication setup because they are co-located, so everything is local and they share all the resources. This is the same thing as execution on the same physical server in a pre-container world.

There are other things to do with pods, of course. You might have an **init container** that initializes a second container. Once the second container is up and serving, the first container stops—its job is done.

Pod model types

There are two model types of pod you can create:

- **One-container-per-pod.** This model is the most popular. The post is the "wrapper" for a single container. Since pod is the smallest object that K8S recognizes, it manages the pods instead of directly managing the containers.
- **Multi-container-pod.** In this model, a pod can hold multiple co-located containers that are tightly coupled to share resources. These containers work as a single, cohesive unit of service. The pod then wraps these multi containers with storage resources into a single unit. Example use cases include sidecars, proxies, logging.

Each pod runs a single instance of your application. If you need to scale the app horizontally (such as running several replicas), you can use a pod per instance. This is different from running multiple containers of the same app within a single pod.

It is worth mentioning that pods are not intended as durable entities. If a node fails or if you're maintaining nodes, the pods won't survive. To solve this issue, K8S has **controllers**—typically, a pod can be created with a type of controller.

Pod lifecycle phases

A pod status tells us where the pod is in its lifecycle. It is meant to give you an idea not for certain, therefore It is good practice to debug if pod does not come up cleanly. The five phases of a pod lifecycle are:

1. **Pending.** The pod is accepted, but at least one container image has not been created.
2. **Running.** The pod is bound to a node, and all containers are created. One container is running or in the process of starting or restarting.
3. **Succeeded.** All containers in the pod successfully terminated and will not restart.
4. **Failed.** All containers are terminated, with at least one container failing. The failed container exited with non-zero status.
5. **Unknown.** The state of the pod couldn't be obtained.

Pods in practice

We have talked about what a pod is in theory, now let's see what it looks like in practice. We'll first go over a simple pod manifest, then we'll deploy an example app showing how to work with it.

The manifest (YAML)

We will break the manifest down into four parts:

- **ApiVersion** – Version of the Kubernetes API you're using
- **Kind** – Kind of object you want to create

- **Metadata** – Information that uniquely identifies the object, such as name or namespace.
- **Spec** – Specified configuration of our pod, for example image name, container name, volumes, etc.

ApiVersion, kind, and metadata are required fields that are applicable to all Kubernetes objects, not just pods. The layout of spec, which is also required, varies across objects. The example manifest shown above shows what a single container pod spec looks like.

```
apiVersion: "api version"              (1)
kind: "object to create"                  (2)
Metadata:                    (3)
  Name: "Pod name"
  labels:
    App: "label value"
Spec:                        (4)
  containers:
  - name: "container name"
    image: "image to use for container"
```

OK, now that we understand how the manifest looks, we'll show both models for creating a pod.

Single container pod

Our pod-1.yaml is the manifest for our single container pod. It runs an nginx pod that echoes something for us.

```
apiVersion: v1
kind: Pod
metadata:
  name: firstpod
  labels:
    app: myapp
spec:
  containers:
  - name: my-first-pod
    image: nginx
```

Next, we deploy this manifest into our local Kubernetes cluster by running **Kubectl create -f pod-1.yaml**. Then we run "kubectl get pods" to confirm that our pod is running as expected.

```
kubectl get pod
NAME                                    READY     STATUS    RESTARTS
AGE
firstpod                                1/1       Running   0
45s
```

It is now running! To confirm it's actually running, run **kubectl exec firstpod—kubeconfig=kubeconfig — service nginx status**. This runs a command inside our pod by passing in **— service nginx status**. (Note: this is similar to running **docker exec**.)

```
kubectl exec firstpod – service nginx status
nginx is running.
```

Now, we'll clean up by running **kubectl delete pod firstpod**.

```
kubectl delete pod firstpod
pod "firstpod" deleted
```

Multi-container manifest

In this example, we will deploy something more useful: a pod with multiple containers that work as a single entity. One container writes the current date to a file every 10 seconds while the other container serves the logs for us.

Go ahead and deploy the pod-2.yaml manifest with **kubectl create -f pod-2.yaml**.

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod # Name of our pod
spec:
  volumes:
  - name: shared-date-logs  # Creating a shared volume for my containers
    emptyDir: {}
  containers:
  - name: container-writing-dates # Name of first container
    image: alpine # Image to use for first container
    command:
    args:  # writing date every 10secs
    volumeMounts:
    - name: shared-date-logs
      mountPath: /var/log # Mounting log dir so app can write to it.
  - name: container-serving-dates # Name of second container
    image: nginx:1.7.9 # Image for second container
    ports:
      - containerPort: 80 # Defining what port to use.
    volumeMounts:
    - name: shared-date-logs
      mountPath: /usr/share/nginx/html # Where nginx will serve the written
file
```

It is worth stopping briefly to touch on **volumes** in pods. In our example above, volumes provide a way for the containers to communicate during the pod's life. If the pod is deleted and recreated, any stored data in the shared volume is lost. **Persistent volumes** object solves this issue so that your data exists beyond pod loss. We are using this multi-container example to not only demonstrate how to create two container pod but to also show the way both containers share resources.

```
kubectl create -f pod-2.yaml
pod "multi-container-pod" created
```

Then we confirm that it's really deployed.

```
kubectl get pod --kubeconfig=kubeconfig
NAME                                        READY     STATUS    RESTARTS
AGE
multi-container-pod                         2/2       Running   0
1m
```

Great! It is running. Now let's make sure things are working as we expect. We need to make sure that our second container is serving the dates.

Check to make sure two containers are in our pod by running **kubectl describe pod "pod name"**. This command shows what the created object looks like.

```
Containers:
  container-writing-dates:
    Container ID:
docker://e5274fb901cf276ed5d94b625b36f240e3ca7f1a89cbe74b3c492347e98c7a5b
    Image:          alpine
    Image ID:       docker-
pullable://alpine@sha256:621c2f39f8133acb8e64023a94dbdf0d5ca81896102b9e57c0dc
184cadaf5528
    Port:
    Host Port:
    Command:
      /bin/sh
    Args:
      -c
      while true; do date >> /var/log/output.txt; sleep 10;done
    State:          Running
      Started:      Fri, 16 Nov 2018 11:31:44 -0700
    Ready:          True
    Restart Count:  0
    Environment:
    Mounts:
      /var/log from shared-date-logs (rw)
      /var/run/secrets/Kubernetes.io/serviceaccount from default-token-8dl5j
(ro)
  container-serving-dates:
    Container ID:
docker://f9c85f3fe398c3197644fb117dc1681635268903b3bba43aa0a1d151fab6ad22
    Image:          nginx:1.7.9
    Image ID:       docker-
pullable://nginx@sha256:e3456c851a152494c3e4ff5fcc26f240206abac0c9d794affb40e
0714846c451
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
```

```
      Started:      Fri, 16 Nov 2018 11:31:44 -0700
    Ready:          True
    Restart Count:  0
    Environment:
    Mounts:
      /usr/share/nginx/html from shared-date-logs (rw)
      /var/run/secrets/Kubernetes.io/serviceaccount from default-token-8dl5j
(ro)
```

Both containers are running, so let's make sure both are doing their assigned jobs.

Connect to the container by running **kubectl exec -ti multi-container-pod -c container-serving-dates —kubeconfig=kubeconfig bash**. Now we are inside the container.

Finally, we run **curl 'http://localhost:80/output.txt'** inside the container and it should serve our file. (If you don't have curl installed in the container, first run **apt-get update && apt-get install curl** then run **curl 'http://localhost:80/output.txt'** again.)

```
curl 'http://localhost:80/app.txt'
Fri Nov 16 18:31:44 UTC 2018
Fri Nov 16 18:31:54 UTC 2018
Fri Nov 16 18:32:04 UTC 2018
Fri Nov 16 18:32:14 UTC 2018
Fri Nov 16 18:32:24 UTC 2018
Fri Nov 16 18:32:34 UTC 2018
Fri Nov 16 18:32:44 UTC 2018
Fri Nov 16 18:32:54 UTC 2018
```

Additional resources

For more on Kubernetes, explore these resources:

- Kubernetes Guide, with 20+ articles and tutorials
- BMC DevOps Blog