

# Kubernetes

## Lab 6 – Volumes, Secrets, and ConfigMaps

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. When a container crashes, the *kubelet* will replace it by rerunning the original image; the files from the dead container will be lost. Also, when running containers together in a pod it is often necessary to share files between those containers. In both cases a volume can be a solution.

A standard Kubernetes volume has an explicit lifetime - the same as the pod that encloses it. This is different from the Docker volume model, wherein volumes remain until explicitly deleted, regardless of whether there are any running containers using it.

Though Kubernetes volumes have the lifespan of the pod, it is important to remember that pods are anchored by the infrastructure container which cannot crash. Thus a pod volume outlives any containers that run within the pod except the *pause* container. Thus volume data is preserved across container restarts. Only when a pod is deleted or the node the pod runs on crashes does the volume cease to exist.

Kubernetes supports many type of volumes, and a pod can use any number of them simultaneously. Volume types supported include:

- `awsElasticBlockStore` – mounts an Amazon Web Service (AWS) EBS volume
- `azureDisk` – mounts a Microsoft Azure Data Disk
- `azureFile` – mounts an Azure File Volume
- `cephfs` - mounts an existing CephFS volume
- `downwardAPI` – mounts pod metadata
- `emptyDir` – provides the pod with an empty directory (data is deleted when the Pod terminates)
- `fibre channel` - allows an existing fibre channel volume to be mounted in a pod
- `flocker` – mounts a flocker volume
- `gcePersistentDisk` – mounts a Google Compute Engine (GCE) block storage volume
- `gitRepo` – mounts an empty directory and clones a git repository into it for your pod to use
- `glusterfs` – mounts a glusterfs volume
- `hostPath` – mounts a given host path into containers using the volume
- `iscsi` – mounts an iSCSI initiator
- `local` (alpha as of 1.7) – local storage device such as a disk, partition or directory
- `nfs` – mounts an NFS share
- `persistentVolumeClaim` – mounts a persistent volume (a cluster defined resource)
- `projected` - maps several existing volume sources into the same directory
- `portworxVolume` - elastic block storage layer that runs hyperconverged with Kubernetes
- `Quobyte` – allows an existing Quobyte volume to be mounted
- `rbd` – mounts a Rados Block Device volume
- `scaleIO` - software-based storage platform of scalable shared block networked storage

- secret – mounts a volume as an in-memory tmpfs (never touches the host disk)
- storageos - allows an existing StorageOS volume to be mounted into your pod
- vsphereVolume – mounts a vSphere VMDK Volume

At its core, a volume is just a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

To use a volume, a pod specifies what volumes to provide for the pod (the *spec.volumes* field) and where to mount those into containers (the *spec.containers.volumeMounts* field.)

A process in a container sees a filesystem view composed from their Docker image and volumes. The Docker image is at the root of the filesystem hierarchy, and any volumes are mounted at the specified paths within the image. Volumes cannot mount onto other volumes or have hard links to other volumes. Each container in the pod must independently specify where to mount each volume.

## 1. Using Volumes

Imagine we have an application assembly which involves two containers. One container runs a Redis cache and the other runs an application that uses the cache. Using a volume to host the Redis data will ensure that if the Redis container crashes, we can have the `kublet` start a brand new copy of the Redis image but hand it the pod volume, preserving the state across crashes.

To simulate this case we'll start a Deployment with a two container pod. One container will be Redis and the other will be BusyBox. We'll mount a shared volume into both containers.

Create a working directory for your project:

```
ubuntu@ip-10-0-2-200:~$ cd ~
ubuntu@ip-10-0-2-200:~$ mkdir vol
ubuntu@ip-10-0-2-200:~$ cd vol
```

Next create the following Deployment config:

```
ubuntu@ip-10-0-2-200:~/vol$ vi vol.yaml
ubuntu@ip-10-0-2-200:~/vol$ cat vol.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
      tier: backend
  template:
    metadata:
```

```

labels:
  app: redis
  tier: backend
spec:
  volumes:
  - name: data
    emptyDir: {}
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - mountPath: /data
      name: data
  - name: shell
    image: busybox
    command: ["tail", "-f", "/dev/null"]
    volumeMounts:
    - mountPath: /shared-master-data
      name: data
ubuntu@ip-10-0-2-200:~/vol$

```

Here our spec creates an emptyDir volume called *data* and then mounts it into both containers. Create the Deployment and then when both containers are running we will exec into them to explore the volume.

First launch the deployment and wait for the pod containers to come up (redis may need to pull from docker hub):

```

ubuntu@ip-10-0-2-200:~/vol$ kubectl create -f vol.yaml

deployment.apps/nginx-deployment created
ubuntu@ip-10-0-2-200:~/vol$

```

```

ubuntu@ip-10-0-2-200:~/vol$ kubectl get deployment,rs,pods

```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.extensions/nginx-deployment	1/1	1	1	9s

  

NAME	DESIRED	CURRENT	READY
replicaset.extensions/nginx-deployment-85665b6fcd	1	1	1

  

NAME	READY	STATUS	RESTARTS	AGE
pod/client	1/1	Running	0	48m
pod/hi	1/1	Running	0	49m
pod/nginx-deployment-85665b6fcd-9972v	2/2	Running	0	22s

```

ubuntu@ip-10-0-2-200:~/vol$

```

Now exec into “shell” and create a file in the shared volume:

```

ubuntu@ip-10-0-2-200:~/vol$ kubectl exec -it -c shell \
$(kubectl get pod -l app=redis -o name | awk -F '/' '{print $2}')
```

```

/ # ls -l

total 40
drwxr-xr-x    2 root    root    12288 Nov  1 22:58 bin
drwxr-xr-x    5 root    root      360 Dec  4 23:50 dev
drwxr-xr-x    1 root    root    4096 Dec  4 23:50 etc
drwxr-xr-x    2 nobody nogroup 4096 Nov  1 22:58 home
dr-xr-xr-x  313 root    root      0 Dec  4 23:50 proc
drwxr-xr-x    1 root    root    4096 Dec  4 23:52 root
drwxrwxrwx    2 999     root    4096 Dec  4 23:50 shared-master-data
dr-xr-xr-x   13 root    root      0 Dec  4 23:50 sys
drwxrwxrwt    2 root    root    4096 Nov  1 22:58 tmp
drwxr-xr-x    3 root    root    4096 Nov  1 22:58 usr
drwxr-xr-x    1 root    root    4096 Dec  4 23:50 var

/ # ls -l /shared-master-data/
total 0

/ # echo "hello shared data" > /shared-master-data/hello.txt

/ # ls -l /shared-master-data/

total 4
-rw-r--r--    1 root    root      18 Dec  4 23:52 hello.txt

/ # exit
ubuntu@ip-10-0-2-200:~/vol$

```

Finally exec into the “redis” container to examine the volume:

```

ubuntu@ip-10-0-2-200:~/vol$ kubectl exec -it -c redis \
$(kubectl get pod -l app=redis -o name | awk -F '/' '{print $2}') -- /bin/sh

# ls -l /data

total 4
-rw-r--r-- 1 root root 18 Dec  4 23:52 hello.txt

# cat /data/hello.txt

hello shared data

# exit
ubuntu@ip-10-0-2-200:~/vol$

```

## 2. Annotations

Kubernetes provides labels for defining selectable metadata on objects. It can also be useful to attach arbitrary non-identifying metadata, for retrieval by API clients, tools and libraries. This information may be large, may be structured or unstructured, may include characters not permitted by labels, etc. Annotations are not used for object selection making it possible for us to ensure that arbitrary metadata does not get picked up by selectors accidentally.

Like labels, annotations are key-value maps listed under the metadata key. Here’s a simple example of a pod spec

including labels and annotation data:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi
  labels:
    zone: us-east-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
    builder: john-doe
...
```

In the next step we'll run a pod with the above metadata and show how to access the metadata from within the pod's containers.

### 3. Downward API Mount

Containers may need to acquire information about themselves. The downward API allows containers to discover information about themselves or the system without the need to call into the Kubernetes cluster.

The Downward API allows configs to expose pod metadata to containers through environment variables or via a volume mount. The downward API volume refreshes its data in step with the `kubelet` refresh loop.

To test the downward API we can create a pod spec that mounts downward api data in the `/dapi` directory. Lots of information can be mounted via the Downward API:

For pods:

- `spec.nodeName`
- `status.hostIP`
- `metadata.name`
- `metadata.namespace`
- `status.podIP`
- `spec.serviceAccountName`
- `metadata.uid`
- `metadata.labels`
- `metadata.annotations`

For containers:

- `requests.cpu`
- `limits.cpu`
- `requests.memory`
- `limits.memory`

This list will likely grow over time. Create the following pod config to demonstrate each of the metadata items in the above list:

```

ubuntu@ip-10-0-2-200:~/vol$ vi dapi.yaml

ubuntu@ip-10-0-2-200:~/vol$ cat dapi.yaml

apiVersion: v1
kind: Pod
metadata:
  name: dapi
  labels:
    zone: us-east-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
    builder: john-doe
spec:
  containers:
    - name: client-container
      image: gcr.io/google_containers/busybox
      command: ["sh", "-c", "tail -f /dev/null"]
      volumeMounts:
        - name: podinfo
          mountPath: /dapi
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
          - path: "name"
            fieldRef:
              fieldPath: metadata.name
          - path: "namespace"
            fieldRef:
              fieldPath: metadata.namespace
ubuntu@ip-10-0-2-200:~/vol$

```

The volume mount hash inside `volumeMounts` within the container spec looks like any other volume mount. The pod volumes list however includes a downwardAPI mount which specifies each of the bits of pod data we want to capture.

To see how this works, run the pod and wait until it's STATUS is Running:

```

ubuntu@ip-10-0-2-200:~$ kubectl create -f dapi.yaml

pod/dapi created
ubuntu@ip-10-0-2-200:~/vol$

```

```

ubuntu@ip-10-0-2-200:~/vol$ kubectl get pod dapi

```

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

```
ubuntu@ip-10-0-2-200:~/vol$
```

Now exec a shell into the pod to display the mounted metadata:

```
ubuntu@ip-10-0-2-200:~$ kubectl exec -it dapi /bin/sh

/ # ls -l /dapi

total 0
lrwxrwxrwx    1 root    root          18 Dec  4 23:54 annotations ->
..data/annotations
lrwxrwxrwx    1 root    root          13 Dec  4 23:54 labels -> ..data/labels
lrwxrwxrwx    1 root    root          11 Dec  4 23:54 name -> ..data/name
lrwxrwxrwx    1 root    root          16 Dec  4 23:54 namespace ->
..data/namespace

/ # cat /dapi/annotations

build="two"
builder="john-doe"
kubernetes.io/config.seen="2018-12-07T04:41:54.789014934Z"
kubernetes.io/config.source="api"

/ # cat /dapi/labels

cluster="test-cluster1"
rack="rack-22"
zone="us-east-coast"

/ # cat /dapi/name

dapi

/ # cat /dapi/namespace

default

/ # exit

ubuntu@ip-10-0-2-200:~/vol$
```

Delete all services, deployments, rss and pods when you are finished exploring.

## 4. Secrets

Secret are Kubernetes objects used to hold sensitive information, such as passwords, OAuth tokens, and SSH keys. Putting this information in a secret is safer and more flexible than putting it verbatim in a pod definition or in a Docker image.

Secrets can be created by Kubernetes and by users. A secret can be used with a pod in two ways:

- Files in a volume mounted on one or more of its containers
- For use by the `kubelet` when pulling images for the pod

Let's test the volume mounted secret approach. First we need to create some secrets. Secrets are objects in Kubernetes just like pods and deployments. Create a config with a list of two secrets (we'll use the Kubernetes List type to support our List of two Secrets.)

```
ubuntu@ip-10-0-2-200:~/vol$ vim secret.yaml
ubuntu@ip-10-0-2-200:~/vol$ cat secret.yaml
apiVersion: v1
kind: List
items:
- kind: Secret
  apiVersion: v1
  metadata:
    name: prod-db-secret
  data:
    password: "dmFsdWUtMg0KDQo="
    username: "dmFsdWUtMQ0K"
- kind: Secret
  apiVersion: v1
  metadata:
    name: test-db-secret
  data:
    password: "dmFsdWUtMg0KDQo="
    username: "dmFsdWUtMQ0K"
ubuntu@ip-10-0-2-200:~/vol$
```

Use “create” to construct your secrets:

```
ubuntu@ip-10-0-2-200:~/vol$ kubectl create -f secret.yaml

secret/prod-db-secret created
secret/test-db-secret created
ubuntu@ip-10-0-2-200:~/vol$
```

Once created you can get and describe Secrets just like any other object:

```
ubuntu@ip-10-0-2-200:~/vol$ kubectl get secret
```

NAME	TYPE	DATA	AGE
default-token-2kmhb	kubernetes.io/service-account-token	3	4h
prod-db-secret	Opaque	2	4s
test-db-secret	Opaque	2	4s

```
ubuntu@ip-10-0-2-200:~/vol$
```

```
ubuntu@ip-10-0-2-200:~/vol$ kubectl describe secret prod-db-secret
```

```
Name:          prod-db-secret
```



```
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:    Opaque

Data
====
password: 11 bytes
username: 9 bytes
ubuntu@ip-10-0-2-200:~/vol$
```

Now we can create and run a pod that uses the secret. The secret will be mounted as a tmpfs volume and will never be written to disk on the node. First create the pod config:

```
ubuntu@ip-10-0-2-200:~/vol$ vim secpod.yaml

ubuntu@ip-10-0-2-200:~/vol$ cat secpod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: prod-db-client-pod
  labels:
    name: prod-db-client
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: prod-db-secret
  containers:
  - name: db-client-container
    image: nginx
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
ubuntu@ip-10-0-2-200:~/vol$
```

Now create the pod.

```
ubuntu@ip-10-0-2-200:~/vol$ kubectl create -f secpod.yaml

pod/prod-db-client-pod created
ubuntu@ip-10-0-2-200:~/vol$
```

```
ubuntu@ip-10-0-2-200:~/vol$ kubectl get pod -l name=prod-db-client

NAME                READY    STATUS    RESTARTS   AGE
prod-db-client-pod  1/1      Running   0           11s
ubuntu@ip-10-0-2-200:~/vol$
```

Now examine your secret volume:

```
ubuntu@ip-10-0-2-200:~/vol$ kubectl exec -it prod-db-client-pod -- ls -l
/etc/secret-volume

total 0
lrwxrwxrwx 1 root root 15 Dec  4 23:55 password -> ../data/password
lrwxrwxrwx 1 root root 15 Dec  4 23:55 username -> ../data/username

ubuntu@ip-10-0-2-200:~/vol$
```

```
ubuntu@ip-10-0-2-200:~/vol$ kubectl exec -it prod-db-client-pod -- cat
/etc/secret-volume/username

value-1

ubuntu@ip-10-0-2-200:~/vol$ kubectl exec -it prod-db-client-pod -- cat
/etc/secret-volume/password

value-2
ubuntu@ip-10-0-2-200:~/vol$
```

Delete only resources you created including all secrets, services, deployments, rrs and pods when you are finished exploring.

## 5. ConfigMaps

Many applications require configuration via some combination of config files, command line arguments, and environment variables. These configuration artifacts should be decoupled from image content in order to keep containerized applications portable. The ConfigMap API resource provides mechanisms to inject containers with configuration data while keeping containers agnostic of Kubernetes. ConfigMap can be used to store fine-grained information like individual properties or coarse-grained information like entire config files or JSON blobs.

There are a number of ways to create a ConfigMap, including via directory upload, file(s), or literal.

### 5.1 Creating a ConfigMap from a directory

We will create a couple of sample property files we will use to populate the ConfigMap.

```
ubuntu@ip-10-0-2-200:~/vol$ cd ~

ubuntu@ip-10-0-2-200:~$ mkdir configmaps

ubuntu@ip-10-0-2-200:~$ cd configmaps/

ubuntu@ip-10-0-2-200:~/configmaps$

ubuntu@ip-10-0-2-200:~/configmaps$ mkdir files
```

```
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ vi ./files/game.properties
ubuntu@ip-10-0-2-200:~/configmaps$ cat ./files/game.properties
```

```
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ vi ./files/ui.properties
ubuntu@ip-10-0-2-200:~/configmaps$ cat ./files/ui.properties
```

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
ubuntu@ip-10-0-2-200:~/configmaps$
```

We will use the `--from-file` option to supply the directory path containing all the properties files.

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl create configmap game-config --from-
file=./files
```

```
configmap/game-config created
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl describe configmaps game-config
```

```
Name:          game-config
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
====
game.properties:
----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

```
ui.properties:
----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice

Events: <none>
ubuntu@ip-10-0-2-200:~/configmaps$
```

## 5.2 Creating ConfigMaps from files

Similar to supplying a directory, we use the `--from-file` switch but specify the files of interest (via multiple flags.)

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl create configmap game-config-2 \
--from-file=./files/ui.properties --from-file=./files/game.properties

configmap/game-config-2 created
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get configmaps game-config-2 -o json
```

```
{
  "apiVersion": "v1",
  "data": {
    "game.properties":
"enemies=aliens\nlives=3\nenemies.cheat=true\nenemies.cheat.level=noGoodRotten\nsecret.code.passphrase=UDDLRBABAS\nsecret.code.allowed=true\nsecret.code.lives=30\n",
    "ui.properties":
"color.good=purple\ncolor.bad=yellow\nallow.textmode=true\nhow.nice.to.look=fairlyNice\n"
  },
  "kind": "ConfigMap",
  "metadata": {
    "creationTimestamp": "2018-03-30T06:09:17Z",
    "name": "game-config-2",
    "namespace": "default",
    "resourceVersion": "128349",
    "selfLink": "/api/v1/namespaces/default/configmaps/game-config-2",
    "uid": "e129a069-33e0-11e8-9062-02ec217beae8"
  }
}
```

```
ubuntu@ip-10-0-2-200:~/configmaps$
```

## 5.3 Override key

Sometimes you don't want to use the file name as the key for this ConfigMap. During its creation we can supply the key as a prefix.

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl create configmap game-config-3 \
--from-file=game-special-key=./files/game.properties

configmap/game-config-3 created
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get configmaps game-config-3 \
-o json | jq .data.\"game-special-key\"

\"enemies=aliens\\nlives=3\\nenemies.cheat=true\\nenemies.cheat.level=noGoodRotten\\ns
ecret.code.passphrase=UDDLRBABAS\\nsecret.code.allowed=true\\nsecret.code.lives=
30\\n\"
ubuntu@ip-10-0-2-200:~/configmaps$
```

## 5.4 Creating ConfigMap from literal values

Unlike the previous methods, with literals we use *--from-literal* and provide the property (key=value.)

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl create configmap special-config \
--from-literal=special.type=charm --from-literal=special.how=very

configmap/special-config created
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get configmaps special-config -o yaml

apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2018-12-07T04:50:23Z
  name: special-config
  namespace: default
  resourceVersion: "61771"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
  uid: 9b6d0727-f9db-11e8-8781-02d9a858fbbc
ubuntu@ip-10-0-2-200:~/configmaps$
```

Delete your ConfigMaps.

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get configmaps | awk '{print $1}' |
sed -e '/NAME/d'
```

```
game-config
game-config-2
game-config-3
special-config
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get configmaps | awk '{print $1}' \
| sed -e '/NAME/d' | xargs kubectl delete configmap
```

```
configmap "game-config" deleted
configmap "game-config-2" deleted
configmap "game-config-3" deleted
configmap "special-config" deleted
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get configmaps
```

```
No resources found.
ubuntu@ip-10-0-2-200:~/configmaps$
```

## 5.5 Consuming a ConfigMap

Like creation, we have a few options on how to consume a ConfigMap including environment variables (DAPI,) command line arguments (DAPI,) and as a Volume.

### 5.5.1 Consume a ConfigMap via environment variables

We will first create a ConfigMap via a spec file. Next we ingest the ConfigMap first in our containers shell environment.

```
ubuntu@ip-10-0-2-200:~/configmaps$ vi env-cm.yaml
ubuntu@ip-10-0-2-200:~/configmaps$ cat env-cm.yaml
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl create -f env-cm.yaml
```

```
configmap/special-config created
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ vi env-pod.yaml
ubuntu@ip-10-0-2-200:~/configmaps$ cat env-pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
  - name: test-container
    image: gcr.io/google_containers/busybox
    command: [ "/bin/sh", "-c", "env" ]
    env:
    - name: SPECIAL_LEVEL_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.how
    - name: SPECIAL_TYPE_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.type
  restartPolicy: Never
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl create -f env-pod.yaml
```

```
pod/dapi-test-pod created
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
dapi	1/1	Running	0	5m
dapi-test-pod	0/1	Completed	0	9s
hello	0/1	Completed	0	1h
nginx-deployment-7c59c95ffb-svq7p	2/2	Running	0	8m
prod-db-client-pod	1/1	Running	0	3m

```
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl logs dapi-test-pod | grep SPECIAL
```

```
SPECIAL_TYPE_KEY=charm
SPECIAL_LEVEL_KEY=very
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl delete pod dapi-test-pod
```

```
pod "dapi-test-pod" deleted
ubuntu@ip-10-0-2-200:~/configmaps$
```

### 5.5.2 Consume a ConfigMap as command line arguments

Using our existing ConfigMap called `special-config`.

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get configmaps

NAME          DATA      AGE
special-config 2          2m
ubuntu@ip-10-0-2-200:~/configmaps$
```

We are now going to use our ConfigMap as part of the container command.

```
ubuntu@ip-10-0-2-200:~/configmaps$ vi cli-pod.yaml
ubuntu@ip-10-0-2-200:~/configmaps$ cat cli-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
  - name: test-container
    image: gcr.io/google_containers/busybox
    command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
    env:
    - name: SPECIAL_LEVEL_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.how
    - name: SPECIAL_TYPE_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.type
  restartPolicy: Never
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl create -f cli-pod.yaml

pod/dapi-test-pod created
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get pods

NAME          READY   STATUS             RESTARTS   AGE
dapi-test-pod 0/1     ContainerCreating  0          4s
```



```
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl logs dapi-test-pod
```

```
very charm
```

```
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl delete pod dapi-test-pod
```

```
pod "dapi-test-pod" deleted
```

```
ubuntu@ip-10-0-2-200:~/configmaps$
```

### 5.5.3 Consume a ConfigMap via a volume

Using existing ConfigMap called `special-config`, we can also mount the ConfigMap.

```
ubuntu@ip-10-0-2-200:~/configmaps$ vi vol-cm.yaml
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ cat vol-cm.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
  - name: test-container
    image: gcr.io/google_containers/busybox
    command: [ "/bin/sh", "-c", "cat /etc/config/special.how" ]
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: special-config
    restartPolicy: Never
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl create -f vol-cm.yaml
```

```
pod/dapi-test-pod created
```

```
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
dapi	1/1	Running	0	7m

```
dapi-test-pod          0/1      Completed    0      9s
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl logs dapi-test-pod
very
ubuntu@ip-10-0-2-200:~/configmaps$
```

```
ubuntu@ip-10-0-2-200:~/configmaps$ kubectl delete pod dapi-test-pod

pod "dapi-test-pod" deleted
ubuntu@ip-10-0-2-200:~/configmaps$
```

## ConfigMap restrictions

ConfigMaps *must* be created before they are consumed in pods. Controllers may be written to tolerate missing configuration data; consult individual components configured via ConfigMap on a case-by-case basis.

ConfigMaps reside in a namespace. They can only be referenced by pods in the same namespace.

Quota for ConfigMap size is a planned feature.

Kubelet only supports use of ConfigMap for pods it gets from the API server. This includes any pods created using kubectl, or indirectly via a replica sets. It does not include pods created via the Kubelet's `--manifest-url` flag, its `--config` flag, or its REST API (these are not common ways to create pods.)

Congratulations you have completed the lab!

*Copyright (c) 2013-2019 RX-M LLC, Cloud Native Consulting, all rights reserved. Licensed for use only by students in Safewrd Ventures class 18-04-19 (Max 10 students)*