RX-M Cloud Native Consulting

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 types of volumes, and a pod can use any number of them simultaneously.

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 `kubelet` 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:

```
user@ubuntu:~$ cd ~

user@ubuntu:~$ mkdir ~/vol && cd ~/vol

user@ubuntu:~/vol$
```

Next create the following Deployment config:

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

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sharing-redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
```

```yaml
      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
```

```
user@ubuntu:~/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):

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

deployment.apps/sharing-redis created

user@ubuntu:~/vol$
```

Check the status of your new resources:

```
user@ubuntu:~/vol$ kubectl get deploy,rs,po

NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/sharing-redis    1/1     1            1           12s

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/sharing-redis-6ccd556555   1         1         1       12s

NAME                                 READY   STATUS    RESTARTS   AGE
pod/sharing-redis-6ccd556555-zrr2v   2/2     Running   0          12s

user@ubuntu:~/vol$
```

When all of your containers are ready, *exec* into “shell” and create a file in the shared volume:

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

/ # ls -l

total 40
drwxr-xr-x    2 root     root         12288 Dec 23 19:21 bin
drwxr-xr-x    5 root     root           360 Jan  8 23:02 dev
drwxr-xr-x    1 root     root          4096 Jan  8 23:02 etc
drwxr-xr-x    2 nobody   nogroup       4096 Dec 23 19:21 home
dr-xr-xr-x  268 root     root             0 Jan  8 23:02 proc
```

```
drwx------    1 root     root          4096 Jan  8 23:03 root
drwxrwxrwx    2 999      root          4096 Jan  8 23:02 shared-master-data
dr-xr-xr-x   13 root     root             0 Jan  8 23:02 sys
drwxrwxrwt    2 root     root          4096 Dec 23 19:21 tmp
drwxr-xr-x    3 root     root          4096 Dec 23 19:21 usr
drwxr-xr-x    1 root     root          4096 Jan  8 23:02 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 Jan  8 23:03 hello.txt

/ # exit

user@ubuntu:~/vol$
```

Finally exec into the "redis" container to examine the volume:

```
user@ubuntu:~/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 Jan  8 23:03 hello.txt

# cat /data/hello.txt

hello shared data

# exit

user@ubuntu:~/vol$
```

By mounting multiple containers inside a pod to the same shared volumes, you can achieve interoperability between those containers. Some useful scenario would be to have a container running a log processor watch an application container's log directory, or have a container prepare a file or configuration before the primary application container starts.

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:

```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
...
```


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:

```
user@ubuntu:~/vol$ nano dapi.yaml && cat dapi.yaml
```

```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
```

```
user@ubuntu:~/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:

```
user@ubuntu:~/vol$ kubectl apply -f dapi.yaml

pod/dapi created

user@ubuntu:~/vol$
```

```
user@ubuntu:~/vol$ kubectl get pod dapi

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

user@ubuntu:~/vol$
```

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

```
user@ubuntu:~/vol$ kubectl exec -it dapi -- /bin/sh

/ # ls -l /dapi

total 0
lrwxrwxrwx    1 root     root            18 Jan  8 23:06 annotations -> ..data/annotations
lrwxrwxrwx    1 root     root            13 Jan  8 23:06 labels -> ..data/labels
lrwxrwxrwx    1 root     root            11 Jan  8 23:06 name -> ..data/name
lrwxrwxrwx    1 root     root            16 Jan  8 23:06 namespace -> ..data/namespace

/ # cat /dapi/annotations

build="two"
builder="john-doe"
kubectl.kubernetes.io/last-applied-configuration="
{\"apiVersion\":\"v1\",\"kind\":\"Pod\",\"metadata\":{\"annotations\":
{\"build\":\"two\",\"builder\":\"john-doe\"},\"labels\":{\"cluster\":\"test-
cluster1\",\"rack\":\"rack-22\",\"zone\":\"us-east-
coast\"},\"name\":\"dapi\",\"namespace\":\"default\"},\"spec\":{\"containers\":[{\"command\":
[\"sh\",\"-c\",\"tail -f
/dev/null\"],\"image\":\"gcr.io/google_containers/busybox\",\"name\":\"client-
container\",\"volumeMounts\":
[{\"mountPath\":\"/dapi\",\"name\":\"podinfo\",\"readOnly\":false}]}],\"volumes\":
[{\"downwardAPI\":{\"items\":[{\"fieldRef\":
{\"fieldPath\":\"metadata.labels\"},\"path\":\"labels\"},{\"fieldRef\":
{\"fieldPath\":\"metadata.annotations\"},\"path\":\"annotations\"},{\"fieldRef\":
{\"fieldPath\":\"metadata.name\"},\"path\":\"name\"},{\"fieldRef\":
{\"fieldPath\":\"metadata.namespace\"},\"path\":\"namespace\"}]},\"name\":\"podinfo\"}]}}\n"
kubernetes.io/config.seen="2020-01-08T15:06:50.25968931-08:00"

/ # cat /dapi/labels

cluster="test-cluster1"
rack="rack-22"
```

```
zone="us-east-coast"

/ # cat /dapi/name

dapi

/ # cat /dapi/namespace

default

/ # exit

user@ubuntu:~/vol$
```

Delete all services, deployments, replicasets and pods when you are finished exploring. For resources created based on files inside a directory, `kubectl delete` can be instructed to delete resources described inside files within a directory.

Delete all the resources created from files inside the ~/vol/ directory:

```
user@ubuntu:~/vol$ kubectl delete -f ~/vol/.

pod "dapi" deleted
deployment.apps "sharing-redis" deleted

user@ubuntu:~/vol$
```

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.)

```
user@ubuntu:~/vol$ nano secret.yaml && cat secret.yaml
```

```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"
```

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

Use “create” to construct your secrets:


```
user@ubuntu:~/vol$ kubectl apply -f secret.yaml

secret/prod-db-secret created
secret/test-db-secret created

user@ubuntu:~/vol$
```

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

```
user@ubuntu:~/vol$ kubectl get secret

NAME                  TYPE                                  DATA   AGE
default-token-7bqf5   kubernetes.io/service-account-token   3      110m
prod-db-secret        Opaque                                2      3s
test-db-secret        Opaque                                2      3s

user@ubuntu:~/vol$
```

```
user@ubuntu:~/vol$ kubectl describe secret prod-db-secret

Name:         prod-db-secret
Namespace:    default
Labels:       <none>
Annotations:
Type:         Opaque

Data
====
password:  11 bytes
username:  9 bytes

user@ubuntu:~/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:

```
user@ubuntu:~/vol$ nano secpod.yaml && cat secpod.yaml
```

```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"
```

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

Now create the pod.

```
user@ubuntu:~/vol$ kubectl apply -f secpod.yaml

pod/prod-db-client-pod created

user@ubuntu:~/vol$ kubectl get pod -l name=prod-db-client

NAME                 READY     STATUS    RESTARTS   AGE
prod-db-client-pod   1/1       Running   0          11s

user@ubuntu:~/vol$
```

Now examine your secret volume:

```
user@ubuntu:~/vol$ kubectl exec prod-db-client-pod -- ls -l /etc/secret-volume

total 0
lrwxrwxrwx 1 root root 15 Jan  8 23:13 password -> ..data/password
lrwxrwxrwx 1 root root 15 Jan  8 23:13 username -> ..data/username

user@ubuntu:~/vol$
```

> N.B. You may notice that the `kubectl exec` commands executed above are a little different than previously shown: the `kubectl` portion describing the pod to act upon is separated from the desired command within the pod with `--`. This approach makes it easier to formulate complex commands to run within a pod's containers, but it is entirely optional.

```
user@ubuntu:~/vol$ kubectl exec prod-db-client-pod -- cat /etc/secret-volume/username

value-1

user@ubuntu:~/vol$ kubectl exec prod-db-client-pod -- cat /etc/secret-volume/password

value-2

user@ubuntu:~/vol$
```

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

```
user@ubuntu:~/vol$ kubectl delete -f ~/vol/.

pod "prod-db-client-pod" deleted
secret "prod-db-secret" deleted
secret "test-db-secret" deleted
Error from server (NotFound): error when deleting "/home/user/vol/dapi.yaml": pods "dapi" not
found
Error from server (NotFound): error when deleting "/home/user/vol/vol.yaml": deployments.apps
"sharing-redis" not found

user@ubuntu:~/vol$
```

> You may have already deleted resources that have specs in the ~/vol directory, hence the "not found" errors.

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.

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

user@ubuntu:~$ mkdir ~/configmaps &&  cd ~/configmaps/

user@ubuntu:~/configmaps$ mkdir files

user@ubuntu:~/configmaps$
```

For the first property file, enter some parameters that would influence a hypothetical game:

```
user@ubuntu:~/configmaps$ nano ./files/game.properties

user@ubuntu:~/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

user@ubuntu:~/configmaps$
```

For the next property file, enter parameters that would influence the hypothetical game's interface:

```
user@ubuntu:~/configmaps$ nano ./files/ui.properties

user@ubuntu:~/configmaps$ cat ./files/ui.properties

color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice

user@ubuntu:~/configmaps$
```

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

```
user@ubuntu:~/configmaps$ kubectl create configmap game-config --from-file=./files

configmap/game-config created

user@ubuntu:~/configmaps$
```

```
user@ubuntu:~/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>

user@ubuntu:~/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):

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

configmap/game-config-2 created

user@ubuntu:~/configmaps$
```

Now check the contents of the game-config-2 configmap you just created from separate files:

```
user@ubuntu:~/configmaps$ kubectl get configmaps game-config-2 -o json
```


```json
{
  "apiVersion": "v1",
  "data": {
      "game.properties":
"enemies=aliens\nlives=3\nenemies.cheat=true\nenemies.cheat.level=noGoodRotten\nsecret.code.passph
rase=UUDDLRLRBABAS\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": "2020-01-08T23:17:20Z",
      "name": "game-config-2",
      "namespace": "default",
      "resourceVersion": "10963",
      "selfLink": "/api/v1/namespaces/default/configmaps/game-config-2",
      "uid": "862803f2-0545-403a-b80e-3b6f0cd46961"
  }
}
```


```
user@ubuntu:~/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.

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

configmap/game-config-3 created

user@ubuntu:~/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\nsecret.code.pass
phrase=UUDDLRLRBABAS\nsecret.code.allowed=true\nsecret.code.lives=30\n"

user@ubuntu:~/configmaps$
```

5.4 Creating ConfigMap from literal values

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

```
user@ubuntu:~/configmaps$ kubectl create configmap special-config \
--from-literal=special.type=charm --from-literal=special.how=very

configmap/special-config created

user@ubuntu:~/configmaps$
```

```
user@ubuntu:~/configmaps$ kubectl get configmaps special-config -o yaml

apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: "2020-01-08T23:19:36Z"
  name: special-config
  namespace: default
  resourceVersion: "11130"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
  uid: d5be3555-9464-4074-87a1-fcfbf57dd71a


user@ubuntu:~/configmaps$
```

Delete your ConfigMaps:

```
user@ubuntu:~/configmaps$ kubectl get configmaps | awk '{print $1}' | sed -e '/NAME/d'

game-config
game-config-2
game-config-3
special-config

user@ubuntu:~/configmaps$
```

```
user@ubuntu:~/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

user@ubuntu:~/configmaps$
```

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

No resources found in default namespace.

user@ubuntu:~/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.

```
user@ubuntu:~/configmaps$ nano env-cm.yaml && cat env-cm.yaml
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

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

```
user@ubuntu:~/configmaps$ kubectl apply -f env-cm.yaml

configmap/special-config created

user@ubuntu:~/configmaps$
```

```
user@ubuntu:~/configmaps$ nano env-pod.yaml && cat env-pod.yaml
```

```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
```

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

This test pod will take the values from the configMap and assign it to environment variables SPECIAL_LEVEL_KEY and SPECIAL_TYPE_KEY in its container. The container itself will run the `env` command to dump any environment variables assigned to it.

```
user@ubuntu:~/configmaps$ kubectl apply -f env-pod.yaml

pod/dapi-test-pod created

user@ubuntu:~/configmaps$
```

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

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

user@ubuntu:~/configmaps$
```

Now, check the container log, grepping for SPECIAL to see if the SPECIAL_LEVEL_KEY and SPECIAL_TYPE_KEY variables were dumped when the container ran the `env` command:

```
user@ubuntu:~/configmaps$ kubectl logs dapi-test-pod | grep SPECIAL

SPECIAL_TYPE_KEY=charm
SPECIAL_LEVEL_KEY=very

user@ubuntu:~/configmaps$
```

Success, the container pulled the level key and type key from the configmap.

Go ahead and remove the dapi test pod:

```
user@ubuntu:~/configmaps$ kubectl delete pod dapi-test-pod

pod "dapi-test-pod" deleted

user@ubuntu:~/configmaps$
```

5.5.2 Consume a ConfigMap as command line arguments

Using our existing ConfigMap called `special-config`.

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

NAME             DATA      AGE
special-config   2         76s

user@ubuntu:~/configmaps$
```

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

```
user@ubuntu:~/configmaps$ nano cli-pod.yaml && cat cli-pod.yaml
```

```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
```

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

Like the dapi-test-pod from the previous step, the container will pull the values of SPECIAL_LEVEL_KEY and SPECIAL_TYPE_KEY from the configmap. This time, however, it will use the container's shell to dump the values of those environment variables.

Create the cli-pod:

```
user@ubuntu:~/configmaps$ kubectl apply -f cli-pod.yaml

pod/dapi-test-pod created

user@ubuntu:~/configmaps$
```

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

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

user@ubuntu:~/configmaps$ kubectl get pods

NAME             READY    STATUS       RESTARTS    AGE
dapi-test-pod    0/1      Completed    0           6s

user@ubuntu:~/configmaps$
```

With the pod created (and completed), check its log to see if the cli command was run and that the environment variables were dumped to its STDOUT:

```
user@ubuntu:~/configmaps$ kubectl logs dapi-test-pod

very charm

user@ubuntu:~/configmaps$
```

Once configMap values are declared as variables, you will be able to consume them as you would any other environment variable inside any pod's container(s).

Remove the dapi-test-pod again:

```
user@ubuntu:~/configmaps$ kubectl delete pod dapi-test-pod

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

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

5.5.3 Consume a ConfigMap via a volume

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

```
user@ubuntu:~/configmaps$ nano vol-cm.yaml && cat vol-cm.yaml
```

```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
```

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

In this spec, the special-config ConfigMap is mounted as a volume on the pod. The volume is then mounted in the pod's container at /etc/config. The container's shell will then read the file special.how that should be mounted there:

```
user@ubuntu:~/configmaps$ kubectl apply -f vol-cm.yaml

pod/dapi-test-pod created

user@ubuntu:~/configmaps$ kubectl get pods

NAME                                    READY     STATUS      RESTARTS    AGE
dapi-test-pod                           0/1       Completed   0           9s

user@ubuntu:~/configmaps$
```

Try to exec into the dapi-test-pod to see how the configMap was mounted:

```
user@ubuntu:~/configmaps$ kubectl exec dapi-test-pod -- /bin/sh -c "ls /etc/config"

error: cannot exec into a container in a completed pod; current phase is Succeeded

user@ubuntu:~/configmaps$
```

Pods in the "completed" status are not actively running their containers, so you will need to check the logs to see if the command succeeded:

```
user@ubuntu:~/configmaps$ kubectl logs dapi-test-pod

very

user@ubuntu:~/configmaps$
```

That worked! When a configMap is mounted in a volume, each key in the volume is treated as a new file that can be found where the configMap was mounted in the pod's container filesystems.

```
user@ubuntu:~/configmaps$ kubectl delete pod dapi-test-pod

pod "dapi-test-pod" deleted

user@ubuntu:~/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.

If ConfigMaps are modified or updated, any pods that use that ConfigMap may need to be restarted in order for the changes made to take effect.

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

Quota for ConfigMap size has not been implemented yet, but etcd does have a 1MB limit for objects stored within it.

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!