rx-m cloud native & ai training & consulting

Kubernetes

Secrets, ConfigMaps, and Downward API

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 images 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.

Secrets 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 container image. All data in a Kubernetes Secrets is encoded in base64, and typically encrypted via an integration with a key management system (KMS).

ConfigMaps and Secrets can be created by Kubernetes and by users. ConfigMaps and Secrets can be used with a pod in three ways:

- Environment variables
- Files in a volume mounted on one or more of its containers
- For use by the kubelet when pulling images for the pod (Secret type only)

0. Prerequisites

If you already know how to login to your lab system and Docker and Kubernetes are installed and configured you can skip this section and jump directly to `1.` . Otherwise, follow the instructions

0.1 Login to your lab system

Login to your machine using an ssh client:

```
@laptop:~$ chmod 400 key.pem

@laptop:~$ ssh -i key.pem ubuntu@55.55.55.55

Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-1031-aws x86_64)

...

~$
```

0.2. Install and configure Docker and Kubernetes

For this lab you will need a Kubernetes cluster. We have prepared a quick installation script that expedites the process of setting up your Lab VM for Kubernetes by performing the following operations:

- Downloading and Installing Docker
- Enabling the apt package manager to retrieve the Kubernetes binaries
- Bootstrapping a Kubernetes Cluster
- Configuring kubectl
- Allowing regular workloads to run on a single node

```
~$ wget -qO - https://raw.githubusercontent.com/RX-
M/classfiles/master/k8s.sh | sh

...

~$
```

We also add our account to the `docker` group to minimize use of `sudo`:

```
~$ sudo usermod -aG docker $(whoami)     # add yourself to the group

~$
```

Add tab completion permanently to your bash shell:

```
~$ echo "source <(kubectl completion bash)" >> ~/.bashrc
```

After updating your user groups and adding tab completion you will need to refresh your shell session. On the lab system the easiest approach is to logout at the command line:

```
~$ exit

logout
Connection to 55.55.55.55 closed.

@laptop:~$
```

Log back in as *ubuntu* with the SSH key:

```
@laptop:~$ ssh -i \path-to-ssh-key\key.pem ubuntu@55.55.55.55

...
```

```
~$
```

Great. Your current shell, and any new shell sessions, can now use the `docker` command without `sudo` and can make use of tab completion.

1. Secrets

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 using `kubectl create` in the following sequence:

```
~$ mkdir ~/appconfig && cd ~/appconfig

~/appconfig$ kubectl create secret generic prod-db-secret --dry-run=client
-o yaml  \
--from-literal username=produser@example.com --from-literal
password=PRod@PW > secret.yaml

~/appconfig$
```

Append `---` to the file using echo and `>>` before the second `create` command to declare separate specs in the manifest):

```
~/appconfig$ echo "---" >> secret.yaml

~/appconfig$
```

Now append the second manifest to the file:

```
~/appconfig$ kubectl create secret generic test-db-secret --dry-run=client
-o yaml \
--from-literal username=testuser@example.com --from-literal
password=TEST#pw >> secret.yaml

~/appconfig$
```

Check the file:

```
~/appconfig$ cat secret.yaml
```

```
apiVersion: v1
data:
  password: UFJvZEBQVw==
  username: cHJvZHVzZXJAZXhhbXBsZS5jb20=
kind: Secret
metadata:
  creationTimestamp: null
  name: prod-db-secret
---
apiVersion: v1
data:
  password: VEVTVCNwdw==
  username: dGVzdHVzZXJAZXhhbXBsZS5jb20=
kind: Secret
metadata:
  creationTimestamp: null
  name: test-db-secret
```

```
~/appconfig$
```

Use `kubectl apply` to create your secrets in the cluster:

```
~/appconfig$ kubectl apply -f secret.yaml

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

~/appconfig$
```

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

```
~/appconfig$ kubectl get secret

NAME             TYPE     DATA   AGE
prod-db-secret   Opaque   2      3s
test-db-secret   Opaque   2      3s

~/appconfig$
```

Note that the values of the secret are obfuscated when using `describe`:

```
~/appconfig$ kubectl describe secret prod-db-secret
```

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

Type:  Opaque

Data
====
password:  7 bytes
username:  20 bytes
```

```
~/appconfig$
```

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:

```
~/appconfig$ kubectl run prod-db-client-pod --image nginx --dry-run=client
-o yaml > secpod.yaml

~/appconfig$
```

Then add the secret as a volume named `secret-volume` to the pod. Mount the `secret-volume` to the container as a readOnly directory at `/etc/secret-volume`:

```
~/appconfig$ nano secpod.yaml && cat $_
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: prod-db-client-pod
  name: prod-db-client-pod
spec:
  containers:
  - image: nginx
    name: prod-db-client-pod
    resources: {}
    volumeMounts:                    # Add this
    - name: secret-volume            # Add this
      readOnly: true                 # Add this
      mountPath: "/etc/secret-volume"  # Add this
  volumes:                           # Add this
```

```
    - name: secret-volume                # Add this
      secret:                            # Add this
        secretName: prod-db-secret       # Add this
    dnsPolicy: ClusterFirst
    restartPolicy: Always
  status: {}
```

```
~/appconfig$
```

Now create the pod:

```
~/appconfig$ kubectl apply -f secpod.yaml

pod/prod-db-client-pod created

~/appconfig$ kubectl get pod prod-db-client-pod

NAME                 READY   STATUS    RESTARTS   AGE
prod-db-client-pod   1/1     Running   0          3s

~/appconfig$
```

Now examine your secret volume:

```
~/appconfig$ kubectl exec prod-db-client-pod -- ls -l /etc/secret-volume

total 0
lrwxrwxrwx 1 root root 15 Jan 18 01:50 password -> ..data/password
lrwxrwxrwx 1 root root 15 Jan 18 01:50 username -> ..data/username

~/appconfig$
```

Echo the values of the files:

> N.B. The `; echo` in each command ensures the `kubectl` output is printed cleanly on a new line.

```
~/appconfig$ kubectl exec prod-db-client-pod -- cat /etc/secret-
volume/username ; echo

produser@example.com

~/appconfig$ kubectl exec prod-db-client-pod -- cat /etc/secret-
volume/password ; echo

PRod@PW
```

```
~/appconfig$
```

Though secrets are encoded (and likely encrypted) by a Kubernetes platform, applications receive the decrypted/decoded values.

Delete the resources you created by passing the path to the configmaps directory to `delete -f`:

```
~/appconfig$ kubectl delete -f ~/appconfig/.

pod "prod-db-client-pod" deleted
secret "prod-db-secret" deleted
secret "test-db-secret" deleted

~/appconfig$
```

2. ConfigMaps

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

2.1. Creating a ConfigMap from a directory

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

```
~/appconfig$ mkdir files

~/appconfig$
```

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

```
~/appconfig$ nano ./files/game.properties && cat $_
```

```
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

```
~/appconfig$
```

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

```
~/appconfig$ nano ./files/ui.properties && cat $_
```

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

```
~/appconfig$
```

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

```
~/appconfig$ kubectl create configmap game-config --from-file=./files

configmap/game-config created

~/appconfig$
```

```
~/appconfig$ 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


BinaryData
====

Events:  <none>
```

```
~/appconfig$
```

Unlike secrets, data stored in configMaps are stored in plain text.

2.2. Creating ConfigMaps from files

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

```
~/appconfig$ kubectl create configmap game-config-2 \
--from-file=./files/ui.properties --from-file=./files/game.properties

configmap/game-config-2 created

~/appconfig$
```

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

```
~/appconfig$ kubectl get configmaps game-config-2 -o json
```

```
{
    "apiVersion": "v1",
    "data": {
        "game.properties":
"enemies=aliens\nlives=3\nenemies.cheat=true\nenemies.cheat.level=noGoodRo
tten\nsecret.code.passphrase=UUDDLRLRBABAS\nsecret.code.allowed=tr\n",
        "ui.properties":
"color.good=purple\ncolor.bad=yellow\nallow.textmode=true\nhow.nice.to.loo
k=fairlyNice\n"
    },
    "kind": "ConfigMap",
    "metadata": {
        "creationTimestamp": "2023-07-26T01:06:44Z",
        "name": "game-config-2",
        "namespace": "default",
        "resourceVersion": "198157",
        "uid": "f0eb7b3c-c9dc-4841-be41-f42b97cae083"
    }
}
```

```
~/appconfig$
```

2.3. Override key

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

```
~/appconfig$ kubectl create configmap game-config-3 \
--from-file=game-special-key=./files/game.properties

configmap/game-config-3 created

~/appconfig$ kubectl get configmaps game-config-3 -o
jsonpath='{.data.game-special-key}'

enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30

~/appconfig$
```

2.4. Creating a ConfigMap from literal values

Unlike the previous methods, with literals we use `--from-literal` and provide the property (key=value.) to create a ConfigMap.

Use the `--from-literal` option to create a configmap using literal statements:

```
~/appconfig$ kubectl create configmap special-config \
--from-literal=special.type=charm --from-literal=special.how=very

configmap/special-config created

~/appconfig$ kubectl get configmap special-config -o yaml
```

```
apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: "2023-07-10T02:01:38Z"
  name: special-config
  namespace: default
  resourceVersion: "3031"
  uid: 6cdbf266-666b-4dd9-ab53-e50b99b0d954
```

```
~/appconfig$
```

Applications that override configurations when new, minimal config files receive the greatest benefit from creating ConfigMaps with `--from-literal`. You can create quick configurations that without having to type out, uncomment, and otherwise modify a potentially long configuration file to affect a change to the application's behavior.

List your current configMaps:

```
~/appconfig$ kubectl get configmaps

NAME                 DATA   AGE
game-config          2      78s
game-config-2        2      36s
game-config-3        1      21s
kube-root-ca.crt     1      29m
special-config       2      14s

~/appconfig$
```

Delete your ConfigMaps using `-o name` with the previous command and the configMap short name `cm`:

```
~/appconfig$ kubectl delete configmap game-config game-config-2 game-
config-3 special-config

configmap "game-config" deleted
configmap "game-config-2" deleted
configmap "game-config-3" deleted
configmap "special-config" deleted

~/appconfig$ kubectl get cm

NAME                 DATA   AGE
kube-root-ca.crt     1      179m
```

```
~/appconfig$
```

2.5. Consuming a ConfigMap

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

2.5.1. Consume a ConfigMap via environment variables

Environment variables are a common method of configuring applications. This is especially true in containerized scenarios, where many applications expose key settings as environment variables. You can create and consume ConfigMaps as environment variables for your pod's containers to take advantage of this.

First create a ConfigMap via a spec file:

```
~/appconfig$ kubectl create configmap special-config --dry-run=client -o
yaml \
--from-literal special.how=very --from-literal special.type=charm -n
default > env-cm.yaml

~/appconfig$ cat env-cm.yaml
```

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

```
~/appconfig$ kubectl apply -f env-cm.yaml

configmap/special-config created

~/appconfig$
```

Next we ingest the ConfigMap first in our container's shell environment. Create the PodSpec structure we will work from:

```
~/appconfig$ kubectl run --dry-run=client -o yaml cm-test-pod --image
busybox \
--restart Never --env SPECIAL_LEVEL_KEY=changeMe --env
SPECIAL_TYPE_KEY=changeMe \
--command -- /bin/sh -c \"env\" > env-pod.yaml

~/appconfig$
```

Edit the environment variables in the manifest to use the ConfigMap as environment variables with the `valueFrom` and `configMapKeyRef` keys under each environment variables:

```
~/appconfig$ nano env-pod.yaml && cat $_
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: cm-test-pod
  name: cm-test-pod
spec:
  containers:
  - command:
    - /bin/sh
    - -c
    - env
    env:
    - name: SPECIAL_LEVEL_KEY
      valueFrom:                # Change this
        configMapKeyRef:        # Add this
          name: special-config  # Add this
          key: special.how      # Add this
    - name: SPECIAL_TYPE_KEY
      valueFrom:                # Change this
        configMapKeyRef:        # Add this
          name: special-config  # Add this
          key: special.type     # Add this
    image: busybox
    name: cm-test-pod
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

```
~/appconfig$
```

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.

```
~/appconfig$ kubectl apply -f env-pod.yaml

pod/cm-test-pod created

~/appconfig$
```

```
~/appconfig$ kubectl get pods

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

~/appconfig$
```

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:

```
~/appconfig$ kubectl logs cm-test-pod | grep SPECIAL

SPECIAL_TYPE_KEY=charm
SPECIAL_LEVEL_KEY=very

~/appconfig$
```

Success! The container pulled the level key and type key from the configmap.

Go ahead and remove the dapi test pod:

```
~/appconfig$ kubectl delete pod cm-test-pod

pod "cm-test-pod" deleted

~/appconfig$
```

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. A container that detects configMap changes and restarts the primary application container in its pod is a good use case for a sidecar!
    - Files projected into a container filesystem from a ConfigMap are read-only and ignore any other settings that affect file permissions on those files (such as `readOnly` in the `spec.containers.volumeMounts` section or the `spec.volumes.configMap.defaultMode`)
    - If a ConfigMap's file contents are updated and they are projected in as files, the files in the pod will reflect the update after about a minute
- 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 ReplicaSets. It does not include pods created via the Kubelet’s `staticPodPath` config, its `--config` flag, or its REST API (these are not common ways to create pods).

3. Downward API

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:

- metadata.name
- metadata.namespace
- metadata.uid
- metadata.labels
- metadata.annotations

For containers:

- requests.cpu
- requests.ephemeral-storage
- requests.memory
- limits.cpu
- limits.ephemeral-storage
- limits.memory

The following is available through environment variables:

- spec.nodeName

- status.hostIP
- status.podIP
- spec.serviceAccountName

This list will likely grow over time and is available here: https://k8s.io/docs/tasks/inject-data-application/downward-api-volume-expose-pod-information/#capabilities-of-the-downward-api. Create the following pod config to demonstrate each of the metadata items in the above list:

```
~/appconfig$ mkdir ~/dapi && cd $_

~/dapi$ kubectl run dapi --dry-run=client -o yaml --image
docker.io/busybox \
-l zone=us-east-coast,cluster=test-cluster1,rack=rack-22 \
--restart Never --command -- /bin/sh -c "tail -f /dev/null" > dapi.yaml

~/dapi$ nano dapi.yaml && cat $_
```

- Add the annotations `build: two` and `builder: john-doe`
- Add the `downwardAPI` with the labels and annotations as shown below as a volume named `podinfo`
- Mount the `podinfo` volume to the client-container at `/dapi`

```
apiVersion: v1
kind: Pod
metadata:
  annotations:                                # Add this
    build: two                                # Add this
    builder: john-doe                         # Add this
  creationTimestamp: null
  labels:
    cluster: test-cluster1
    rack: rack-22
    zone: us-east-coast
  name: dapi
spec:
  containers:
  - command:
    - /bin/sh
    - -c
    - tail -f /dev/null
    image: docker.io/busybox
    name: dapi
    resources: {}
    volumeMounts:                             # Add this
    - name: podinfo                           # Add this
      mountPath: /dapi                        # Add this
      readOnly: false                         # Add this
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  volumes:                                    # Add this
  - name: podinfo                             # Add this
```

```
    downwardAPI:                            # Add this
      items:                                # Add this
      - path: "labels"                      # Add this
        fieldRef:                           # Add this
          fieldPath: metadata.labels        # Add this
      - path: "annotations"                 # Add this
        fieldRef:                           # Add this
          fieldPath: metadata.annotations   # Add this
status: {}
```

```
~/dapi$
```

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.

We mounted the pod's annotations as one of the volumes. Annotations store arbitrary non-identifying metadata for Kubernetes objects used for retrieval by API clients, tools and libraries. This information may vary in size, structure, or may include characters not permitted by labels, etc. Annotations are not used for object selection to ensure that arbitrary metadata does not get picked up by Kubernetes selectors accidentally.

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

```
~/dapi$ kubectl apply -f dapi.yaml

pod/dapi created

~/dapi$ kubectl get po dapi

NAME   READY   STATUS    RESTARTS   AGE
dapi   1/1     Running   0          3s

~/dapi$
```

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

```
~/dapi$ kubectl exec -it dapi -- ls -l /dapi

total 0
lrwxrwxrwx    1 root     root            18 Jan 18 01:54 annotations ->
..data/annotations
lrwxrwxrwx    1 root     root            13 Jan 18 01:54 labels ->
..data/labels

~/dapi$ kubectl exec -it dapi -- 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\"},\"creationTimestamp\":null,\"labels\":{\"cluster\":\"test-
cluster1\",\"rack\":\"rack-22\",\"zone\":\"us-east-
coast\"},\"name\":\"dapi\",\"namespace\":\"default\"},\"spec\":
{\"containers\":[{\"command\":[\"/bin/sh\",\"-c\",\"tail -f
/dev/null\"],\"image\":\"busybox\",\"name\":\"dapi\",\"resources\":
{},\"volumeMounts\":
[{\"mountPath\":\"/dapi\",\"name\":\"podinfo\",\"readOnly\":false}]}],\"dn
sPolicy\":\"ClusterFirst\",\"restartPolicy\":\"Never\",\"volumes\":
[{\"downwardAPI\":{\"items\":[{\"fieldRef\":
{\"fieldPath\":\"metadata.labels\"},\"path\":\"labels\"},{\"fieldRef\":
{\"fieldPath\":\"metadata.annotations\"},\"path\":\"annotations\"}]},\"nam
e\":\"podinfo\"}]},\"status\":{}}\n"
kubernetes.io/config.seen="2023-07-10T02:03:14.491165170Z"
kubernetes.io/config.source="api"

~/dapi$
```

We see the contents of the annotations, which we didn't include in our original spec. Kubernetes stores some miscellaneous information, like previous configurations or commands that changed the resource as annotations, which is what you see.

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 ~/dapi/ directory:

```
~/dapi$ kubectl delete -f ~/dapi/

pod "dapi" deleted

~/dapi$
```

CHALLENGES

Try the following exercises on your own, using the class slides and the knowledge from this lab to guide you.

First, recreate the `special-config` ConfigMap if you deleted it:

```
~/dapi$ cd ~/appconfig

~/appconfig$ kubectl delete configmap special-config
```

```
~/appconfig$ kubectl create configmap special-config \
--from-literal=special.type=charm --from-literal=special.how=very

configmap/special-config created

~/appconfig$
```

CHALLENGE 1

Mount the values of the `special-config` configmap as environment variables `SPECIAL_LEVEL_KEY` and `SPECIAL_TYPE_KEY`, and modify the `dapi-test-pod` to output the value of those variables to stdout

- Configure the pod with the command `/bin/sh -c "echo $SPECIAL_LEVEL_KEY $SPECIAL_TYPE_KEY"`

CHALLENGE 2

Modify the dapi-test-pod manifest to mount the `special-config` configmap as a volume

- Configure the pod with the command `/bin/sh -c "cat /etc/config/special.how"`
- You should be able to use `kubectl logs dapi-test-pod` to verify that the contents of special.how are printed
- The keys used to define a configMap as a volume may be different

Delete any resources you created after you complete the challenges.

CHALLENGE 3

Deploy a pod running the `httpd:2.4` image that uses the following `index.html` document:

```
~/appconfig$ nano index.html && cat $_
```

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello!</title>
    </head>
    <body>
    <p>The page has loaded successfully!</p>
    </body>
</html>
```

- The index.html should be stored in the API and supplied to the pod's container when the pod is created.
- Webpages for Apache are placed in the `/usr/local/apache2/htdocs/` directory

Congratulations you have completed the lab!