Lab 9 – ConfigMaps and Secrets

Introduction

1. Configure Liveness and Readiness Probes

1. The kubelet uses **liveness** probes to know when to restart a Container. liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

2. The kubelet uses **readiness** probes to know when a Container is ready to start accepting traffic. A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

Define a liveness command

Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted. Kubernetes provides liveness probes to detect and remedy such situations.

1. Create a Pod that runs a Container based on the k8s.gcr.io/busybox image and name it as “**exec-liveness.yaml**“:

Copy

```
cat > exec-liveness.yaml <<EOF

apiVersion: v1

kind: Pod

metadata:

  labels:

    test: liveness

  name: liveness-exec

spec:
```

```
  containers:

  - name: liveness

    image: k8s.gcr.io/busybox

    args:

    - /bin/sh

    - -c

    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600

    livenessProbe:

      exec:

        command:

        - cat

        - /tmp/healthy

      initialDelaySeconds: 5

      periodSeconds: 5

EOF
```

In the configuration file, you can see that the Pod has a single Container.

- The **periodSeconds** field specifies that the kubelet should perform a liveness probe every 5 seconds.
- The **initialDelaySeconds** field tells the kubelet that it should wait 5 second before performing the first probe.

To perform a probe, the kubelet executes the command `cat /tmp/healthy` in the Container.

- If the command succeeds, it **returns 0**, and the kubelet considers the Container to be alive and healthy.

- If the command **returns a non-zero value**, the kubelet kills the Container and restarts it.

2. When the Container starts, it executes this command:

/bin/sh -c “touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600”

For the first 30 seconds of the Container’s life, there is a /**tmp**/**healthy** file. So during the first 30 seconds, the command `cat /tmp/healthy` returns a success code. After 30 seconds, `cat /tmp/healthy` returns a failure code.

3. Create the Pod:

Copy

```
kubectl create -f exec-liveness.yaml
```

Copy

```
kubectl get pods
```

Output:

```
NAME            READY     STATUS    RESTARTS   AGE

liveness-exec   1/1       Running   67         5h

liveness-http   1/1       Running   76         5h

nginx           1/1       Running   0          5h
```

Within 30 seconds, view the Pod events:

Copy

```
kubectl describe pod liveness-exec
```

The output indicates that no liveness probes have failed yet:

```
FirstSeen     LastSeen     Count   From            SubobjectPath
Type         Reason       Message
```

```
--------- --------    -----   ----            -------------
--------    ------      -------

24s       24s     1   {default-scheduler }                    Normal
Scheduled   Successfully assigned liveness-exec to worker0

23s       23s     1   {kubelet worker0}   spec.containers{liveness}
Normal      Pulling     pulling image "k8s.gcr.io/busybox"

23s       23s     1   {kubelet worker0}   spec.containers{liveness}
Normal      Pulled      Successfully pulled image "k8s.gcr.io/busybox"

23s       23s     1   {kubelet worker0}   spec.containers{liveness}
Normal      Created     Created container with docker id 86849c15382e;
Security:[seccomp=unconfined]

23s       23s     1   {kubelet worker0}   spec.containers{liveness}
Normal      Started     Started container with docker id 86849c15382e
```

After 35 seconds, view the Pod events again:

Copy

```
kubectl describe pod liveness-exec
```

At the bottom of the output, there are messages indicating that the liveness probes have failed, and the containers have been killed and recreated.

```
FirstSeen LastSeen    Count   From            SubobjectPath
Type        Reason      Message

--------- --------    -----   ----            -------------
--------    ------      -------

37s       37s     1   {default-scheduler }                    Normal
Scheduled   Successfully assigned liveness-exec to worker0

36s       36s     1   {kubelet worker0}   spec.containers{liveness}
Normal      Pulling     pulling image "k8s.gcr.io/busybox"

36s       36s     1   {kubelet worker0}   spec.containers{liveness}
Normal      Pulled      Successfully pulled image "k8s.gcr.io/busybox"
```

```
36s       36s     1   {kubelet worker0}   spec.containers{liveness}
Normal      Created     Created container with docker id 86849c15382e;
Security:[seccomp=unconfined]

36s       36s     1   {kubelet worker0}   spec.containers{liveness}
Normal      Started     Started container with docker id 86849c15382e

2s        2s      1   {kubelet worker0}   spec.containers{liveness}
Warning     Unhealthy   Liveness probe failed: cat: can't open '/tmp/h
ealthy': No such file or directory
```

Wait another 30 seconds, and verify that the Container has been restarted:

Copy

```
kubectl get pod liveness-exec
```

The output shows that RESTARTS has been incremented:

```
NAME            READY     STATUS    RESTARTS   AGE

liveness-exec   1/1       Running   1          1m
```

Define a liveness HTTP request
Another kind of liveness probe uses an HTTP GET request.

1. Create a Pod that runs a container based on the k8s.gcr.io/liveness image and name it as "**http-liveness.yaml**":

Copy

```
cat > http-liveness.yaml <<EOF

apiVersion: v1

kind: Pod

metadata:

  labels:
```

```
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
EOF
```

In the configuration file, you can see that the Pod has a **single** Container.

- The **periodSeconds** field specifies that the kubelet should perform a liveness probe every 3 seconds.
- The **initialDelaySeconds** field tells the kubelet that it should wait 3 seconds before performing the first probe.

To perform a probe, the kubelet sends an HTTP GET request to the server that is running in the Container and listening on port 8080.

- If the handler for the server's /healthz path returns a success code, the kubelet considers the Container to be alive and healthy.
- If the handler returns a failure code, the kubelet kills the Container and restarts it.

Any code greater than or equal to 200 and less than 400 indicates success. Any other code indicates failure.

Create the source code for the server in **server.go**.

Copy

```
cat > server.go <<EOF

package main



import (

       "fmt"

       "log"

       "net/http"

       "time"

)



func main() {

       started := time.Now()

       http.HandleFunc("/started", func(w http.ResponseWriter, r *http
.Request) {

              w.WriteHeader(200)

              data := (time.Now().Sub(started)).String()
```

```
                w.Write([]byte(data))

        })

        http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http
.Request) {

                duration := time.Now().Sub(started)

                if duration.Seconds() > 10 {

                        w.WriteHeader(500)

                        w.Write([]byte(fmt.Sprintf("error: %v", duration.S
econds())))

                } else {

                        w.WriteHeader(200)

                        w.Write([]byte("ok"))

                }

        })

        log.Fatal(http.ListenAndServe(":8080", nil))

}

EOF
```

For the first 10 seconds that the Container is alive, the /healthz handler returns a status of 200. After that, the handler returns a status of 500.

To try the HTTP liveness check, create a Pod:

Copy

```
kubectl create -f http-liveness.yaml
```

After 10 seconds, view Pod events to verify that liveness probes have failed and the Container has been restarted:

Copy

```
kubectl describe pod liveness-http
```

Define a TCP liveness probe
A third type of liveness probe uses a TCP Socket. With this configuration, the kubelet will attempt to open a socket to your container on the specified port. If it can establish a connection, the container is considered healthy, if it can't it is considered a failure.

Copy

```
cat > tcp-liveness-readiness.yaml <<EOF

apiVersion: v1

kind: Pod

metadata:

  name: goproxy

  labels:

    app: goproxy

spec:

  containers:

  - name: goproxy

    image: k8s.gcr.io/goproxy:0.1

    ports:

    - containerPort: 8080

    readinessProbe:

      tcpSocket:
```

```
        port: 8080

      initialDelaySeconds: 5

      periodSeconds: 10

    livenessProbe:

      tcpSocket:

        port: 8080

      initialDelaySeconds: 15

      periodSeconds: 20

EOF
```

As you can see, configuration for a TCP check is quite similar to an HTTP check. This example uses both readiness and liveness probes. The kubelet will send the first readiness probe 5 seconds after the container starts. This will attempt to connect to the goproxy container on port 8080. If the probe succeeds, the pod will be marked as ready. The kubelet will continue to run this check every 10 seconds.

2. Assign Pods to Nodes

Assigning a Kubernetes Pod to a particular node in a Kubernetes cluster.

a. Add a label to a node

1. List the nodes in your cluster:

Copy

```
kubectl get nodes
```

The output is similar to this:

```
NAME                          STATUS    ROLES     AGE       VERSION

pod40-master.onecloud.com     Ready     master    1d        v1.10.0
```

```
pod40-node.onecloud.com     Ready     node      1d        v1.10.0
```

2. Chose one of your nodes, and add a label to it:

Copy

```
kubectl label nodes pod-master disktype=ssd
```

Output:

```
node "pod40-master.onecloud.com" labeled
```

Verify that your chosen node has a **disktype=ssd** label:

Copy

```
kubectl get nodes --show-labels
```

The output is similar to this:

```
NAME      STATUS    AGE     VERSION            LABELS

 worker0   Ready     1d      v1.6.0+fff5156     ...,disktype=ssd,kuber
netes.io/hostname=worker0

 worker1   Ready     1d      v1.6.0+fff5156     ...,kubernetes.io/host
name=worker1

 worker2   Ready     1d      v1.6.0+fff5156     ...,kubernetes.io/host
name=worker2
```

In the preceding output, you can see that the **worker0** node has a **disktype=ssd** label.

b. Create a pod that gets scheduled to your chosen node

This pod configuration file describes a pod that has a node selector, **disktype: ssd**.
This means that the pod will get scheduled on a node that has a disktype=ssd label.

Copy

```
cat > pod.yaml <<EOF

apiVersion: v1

kind: Pod

metadata:

  name: nginx

  labels:

    env: test

spec:

  containers:

  - name: nginx

    image: nginx

    imagePullPolicy: IfNotPresent

  nodeSelector:

    disktype: ssd

EOF
```

Use the configuration file to create a pod that will get scheduled on your chosen node:

Copy

```
kubectl create -f pod.yaml
```

Verify that the pod is running on your chosen node:

Copy

```
kubectl get pods --output=wide
```

output :

```
NAME      READY      STATUS     RESTARTS    AGE     IP            NODE

 nginx     1/1        Running    0           13s     10.200.0.4    worker0
```

3. Configure a Pod to Use a ConfigMap

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. This section provides a series of usage examples demonstrating how to create ConfigMaps and configure Pods using data stored in ConfigMaps.

Create a ConfigMap

Use the `kubectl create configmap` command to create configmaps from **directories**, **files**, or **literal** values:

Syntax:

```
kubectl create configmap <map-name> <data-source>
```

where <map-name> is the name you want to assign to the ConfigMap <data-source>is the directory, file, or literal value to draw the data from.

The data source corresponds to a key-value pair in the ConfigMap, where

- **key** = the file name or the key you provided on the command line
- **value** = the file contents or the literal value you provided on the command line.

a. Create ConfigMaps from directories

Consider a directory with some files that already contain the data with which you want to populate a ConfigMap:

Copy

```
mkdir example-files
```

Copy

```
cat > example-files/game.properties <<EOF

enemies=aliens

lives=3

enemies.cheat=true

enemies.cheat.level=noGoodRotten

secret.code.passphrase=UUDDLRLRBABAS

secret.code.allowed=true

secret.code.lives=30

EOF
```

Copy

```
cat > example-files/ui.properties <<EOF

color.good=purple

color.bad=yellow

allow.textmode=true

how.nice.to.look=fairlyNice

EOF
```

Copy

```
ls example-files
```

Output:

```
game.properties
```

```
ui.properties
```

Copy

```
ubectl create configmap game-config --from-file=example-files
```

Output:

```
configmap "game-config" created
```

Copy

```
kubectl describe configmaps game-config
```

Copy

```
kubectl get configmaps game-config -o yaml
```

Output:

```
apiVersion: v1

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

kind: ConfigMap

metadata:

  creationTimestamp: 2016-02-18T18:34:05Z

  name: game-config

  namespace: default

  resourceVersion: "407"-

  selflink: /api/v1/namespaces/default/configmaps/game-config

  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

2. Creating configMaps from files

Copy

```
kubectl create configmap game-config-2 --from-file=example-files/game.
properties --from-file=example-files/ui.properties
```

Copy

```
kubectl describe configmaps game-config-2
```

Copy

```
kubectl get configmaps game-config-2 -o yaml
```

3. Environment Variable

Copy

```
cat > example-files/game-env-file.properties <<EOF

enemies=aliens

lives=3

allowed="true"

EOF
```

Copy

```
kubectl create configmap game-config-env-file --from-env-file=example-files/game-env-file.properties
```

Output:

```
configmap "game-config-env-file" created
```

Copy

```
kubectl get configmap game-config-env-file -o yaml
```

Output:

```
apiVersion: v1

data:

  allowed: '"true"'

  enemies: aliens

  lives: "3"

kind: ConfigMap
```

```
metadata:

  creationTimestamp: 2018-04-13T13:21:00Z

  name: game-config-env-file

  namespace: default

  resourceVersion: "127294"

  selfLink: /api/v1/namespaces/default/configmaps/game-config-env-file

  uid: 825bb58e-3f1d-11e8-8f41-fa163e7e98e4
```

5. Create ConfigMaps from literal values

Copy

```
kubectl create configmap special-config --from-literal=special.how=ver
y --from-literal=special.type=charm
```

Copy

```
kubectl get configmaps special-config -o yaml
```

Output:

```
apiVersion: v1

data:

  special.how: very

  special.type: charm

kind: ConfigMap

metadata:

  creationTimestamp: 2018-04-13T13:26:43Z
```

```
  name: special-config

  namespace: default

  resourceVersion: "127713"

  selfLink: /api/v1/namespaces/default/configmaps/special-config

  uid: 4e79d3d5-3f1e-11e8-8f41-fa163e7e98e4
```

Managing Secrets

Objects of type **secret** are intended 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.

Creating your own Secrets

The username and password that the pods should use is in the files ./username.txt and ./password.txt on your local machine.

Copy

```
echo -n "admin" > ./username.txt

echo -n "sjc-0801" > ./password.txt
```

Copy

```
kubectl create secret generic db-user-pass --from-file=./username.txt
--from-file=./password.txt
```

Output:

```
secret "db-user-pass" created
```

You can check that the secret was created like this:

Copy

```
kubectl get secrets
```

Output:

```
NAME                  TYPE                                  DATA
AGE

db-user-pass          Opaque                                2
51s
```

Copy

```
kubectl describe secrets/db-user-pass
```

Output:

```
Name:            db-user-pass

Namespace:       default

Labels:

Annotations:



Type:            Opaque



Data

====

password.txt:    12 bytes

username.txt:    5 bytes
```

Note that neither get nor describe shows the contents of the file by default. This is to protect the secret from being exposed accidentally to someone looking or from being stored in a terminal log.

Creating a Secret Manually

You can also create a secret object in a file first, in json or yaml format, and then create that object.

Each item must be base64 encoded:

Copy

```
echo -n "admin" | base64
```

Output:

```
YWRtaW4=
```

Copy

```
echo -n "sjc-0801" | base64
```

Output:

```
c2pjLTA4MDE=
```

Now write a secret object that looks like this:

Copy

```
cat > ./secret.yaml <<EOF

apiVersion: v1

kind: Secret

metadata:

  name: mysecret
```

```
type: Opaque

data:

  username: YWRtaW4=

  password: c2pjLTA4MDE=

EOF
```

The data field is a map. Its keys must consist of alphanumeric characters, ‘-‘, ‘_’ or ‘.’. The values are arbitrary data, encoded using base64.

Create the secret using kubectl create:

Copy

```
kubectl create -f ./secret.yaml
```

Output:

```
secret "mysecret" created
```

Decoding a Secret

Secrets can be retrieved via the kubectl get secret command. For example, to retrieve the secret created in the previous section:

Copy

```
kubectl get secret mysecret -o yaml
```

Output:

```
apiVersion: v1

data:

  username: YWRtaW4=
```

```
  password: c2pjLTA4MDE=
kind: Secret
metadata:
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
```

Decode the password field:

Copy

```
echo "c2pjLTA4MDE=" | base64 --decode
```

Output:

```
sjc-0801
```