7: Resource Organization¶

Environment Setup¶

Let's get your lab environment setup by automatically performing necessary steps from previous labs in this course. We've automated it for you with the following command:

```
eval "$BOOTSTRAP_COMMAND"
```

7.1: Exploring Namespaces¶

Namespaces enable you to isolate objects within the same Kubernetes cluster. The isolation comes in the form of:

- Object & DNS Name Scoping
- Object Access Control
- Resource Quotas

Step 1: Current and secondary namespaces¶

In this lab you will be working with two namespaces. In this environment, `kubectl` is configured to target the namespace `$SESSION_NAMESPACE`.

```
echo $SESSION_NAMESPACE

# Do you see your session namespace in the output of the following command?
kubectl get namespaces
```

A second namespace has been created for you, its name is `my-namespace`, prefixed with your session namespace name for uniqueness. This second namespace has no running pods at the moment.

Step 2: List resources in Namespaces¶

Many objects in Kubernetes are created within a Namespace. By default, if no namespace is specified when creating a namespaced object, it will be placed in a namespace associated with your current configuration context.

Let's verify this by listing the pods in a few namespaces.

```
# There should be no pods in this namespace since we just created it
kubectl get pods --namespace "${SESSION_NAMESPACE}-my-namespace"

# All of the gowebapp pods we created are in the "current" session namespace
kubectl get pods --namespace $SESSION_NAMESPACE

# Since the session namespace is our current namespace, the output of this command
# should be the same as the last
kubectl get pods
```

7.2: DNS Namespacing¶

Namespaces also impact DNS resolution within the cluster. Pods can connect to services in their own namespace using the service's short name. If a pod needs to connect to a service in a different Namespace, it must use the service's fully-qualified name.

```
*** EXAMPLE OUTPUT ****

$ kubectl get services

NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
gowebapp         ClusterIP   10.104.159.241   <none>        8080/TCP   8m11s
gowebapp-mysql   ClusterIP   10.100.107.166   <none>        3306/TCP   8m22s
```

Let's test connecting to the `gowebapp` service using curl from within a pod three different ways:

- using the short name `gowebapp` from the same namespace that the gowebapp service is deployed to (the session namespace)

- using the short name `gowebapp` from a different namespace than the one that the gowebapp service is deployed to (this should fail)
- using the fully-qualified name `gowebapp.$SESSION_NAMESPACE.svc.cluster.local`

The easiest way to do this is to use a command like the following, which runs the `curl` command to access the service using its short name, in the session namespace:

```
# Run curl in the session namespace, to the short name
kubectl run curl --namespace $SESSION_NAMESPACE --image=curlimages/curl -i --tty --rm
--restart Never -- curl gowebapp:8080
```

Use variations of the above command to launch pods in the desired namespaces. Then try curling the short and fully-qualified URLs above. Be sure to append the port number that the service listening on, which is `8080`.

If successful you should see a response like the following

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">


    <meta name="description" content="">
    <meta name="keywords" content="">
    <meta name="author" content="">

    <title>Welcome</title>

*** OUTPUT TRUNCATED ***
```

Attempting to access a service in a different namespace without using its fully-qualified name should result in an error.

7.3: Changing Default Namespace¶

Right now, if we run any `kubectl` commands without specifying a namespace, the namespace $SESSION_NAMESPACE will be used. If desired, we can set a different namespace to be our default.

Step 01: view the kubectl configuration file¶

The `kubectl` configuration file, also called `kubeconfig` can be viewed in two ways.

```
cat ~/.kube/config

# or

kubectl config view
```

```
# EXAMPLE OUTPUT
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    server: https://172.20.0.1:443
  name: educates
contexts:
- context:
    cluster: educates
    namespace: kubeacademy-w17-s019
    user: educates
  name: educates
current-context: educates
kind: Config
preferences: {}
users:
- name: educates
  user:
    token: REDACTED
```

You'll notice that there are three sections:

- A cluster contains endpoint data for a kubernetes cluster.
- A user defines client credentials for authenticating to a kubernetes cluster.

- A context defines a named cluster, user, namespace tuple which is used to send requests to the specified cluster using the provided authentication info and namespace.

Step 02: create a new context¶

In addition to mapping users to clusters, contexts are also where we can set a default namespace. Let's create a new context that uses the existing user and cluster, but defaults to the `${SESSION_NAMESPACE}-my-namespace` namespace.

```
kubectl config set-context my-context --user educates --cluster educates --namespace
${SESSION_NAMESPACE}-my-namespace
```

Step 03: view the changes to kubeconfig¶

```
kubectl config view
```

You should see the new `my-context` context which includes `my-namespace` as its default namespace.

Step 04: use the new context¶

Next, we need to tell `kubectl` to use our new context.

```
kubectl config use-context my-context
```

At any time, you can see which context is currently active by running

```
kubectl config current-context
```

Or see a list of available contexts by running

```
kubectl config get-contexts
```

Step 05: deploy to my-namespace¶

Let's deploy a test deployment to `my-namespace`. Since it is now our default namespace, we shouldn't need to specify it manually.

```
kubectl create deployment nginx --image=bitnami/nginx

kubectl scale --replicas=3 deployment nginx
```

Step 06: list the deployment and its pods¶

```
kubectl get deployments

kubectl get pods
```

Step 07: list the gowebapp deployment and its pods¶

We can still list the gowebapp deployment and pods, but we'll need to tell kubectl to target the session namespace.

```
kubectl get deployments --namespace $SESSION_NAMESPACE

# -n is short for --namespace
kubectl get pods -n $SESSION_NAMESPACE
```

Step 08: cleanup¶

Before continuing we'll need to clean up a few things.

Delete the nginx deployment

```
kubectl delete deployment nginx
```

Switch back to the default context

```
kubectl config use-context educates
```

kubectl¶

Controlling Output¶

Let's create a quick deployment for testing output and interaction:

```
kubectl create deployment nginx --image=bitnami/nginx

kubectl scale --replicas=3 deployment nginx
```

Explore the various output of resources by trying the different output options. For example:

```
kubectl get deployment nginx -o [json|yaml|wide]
```

A useful output format is to filter the raw json output through jsonpath. This is handy for piping the output of a `kubectl get` command directly to other tools like sed, awk, grep, etc.

For example, let's figure out the command to see how we can list each namespace name by itself newline separated. Use this reference guide for help: https://kubernetes.io/docs/reference/kubectl/jsonpath

Pod Interaction¶

Now that we've played around a bit with kubectl and output, let's explore some of the commands that interact with pods and containers. The goal is the following:

- Forward the nginx port 8080 from one of the pods onto localhost:8000 of your client machine
- Start a tail with follow of the logs on the same pod used above

- Using curl http://localhost:8000, cause some activity on the nginx server using the local port
- Run a remote command inside the container to for example get the current date and time

kubectl Filtering with Labels¶

We can also filter queries with `kubectl` using labels.

For instance, we can list pods with the label `tier=frontend`.

```
kubectl get pods -l tier=frontend
```

In our small cluster with only a few pods and deployments, filtering using labels won't provide much benefit. In large production environments however, this feature is crucial for locating specific resources amongst hundreds or even thousands of pods and deployments.

Cleaning Up¶

Let's clean up the nginx deployment. Since we did not put nginx in its own namespace this time, we cannot simply delete the entire namespace to clean all. We must specifically clean up just the nginx deployment.

```
kubectl delete deployment nginx
```