Deploying Applications on Kubernetes Clusters

Once you have created Kubernetes cluster using the Cisco Container Platform web interface, you can deploy containerized applications on top of it.

This chapter contains the following topics:

Workflow of Deploying Applications

Task	Related Section
Create Kubernetes clusters using the Cisco Container Platform web interface.	Creating Kubernetes Clusters
Download the kubeconfig file that contains the cluster information and the certificates required to access clusters.	Downloading Kubeconfig File, on page 1
Use the kubectl utility to deploy the application and test the scenario.	Sample Scenarios, on page 2

Downloading Kubeconfig File

You must download the cluster environment to access the Kubernetes clusters using command line tools such as `kubectl` or using APIs.

Step 1 From the left pane, click **Clusters**.

Step 2 Click the **Download** icon corresponding to the cluster environment that you want to download.

The `kubeconfig` file that contains the cluster information and the certificates required to access clusters is downloaded to your local system.

Sample Scenarios

This topic contains a few sample scenarios of deploying applications.

Deploying a Pod with Persistent Volume

Tenant Clusters are deployed with a default storage class named `standard`. Depending on the storage class that you have selected during the cluster creation task, persistent volume is made available by a HyperFlex or a vSphere provider.

Step 1 Configure a tenant Kubernetes cluster.

```
export KUBECONFIG=<Path to kubeconfig file>
```

Step 2 Verify if the storage cluster is created.

```
$ kubectl describe storageclass standard

      Name:                  standard
  IsDefaultClass:        Yes
  Annotations:           storageclass.beta.kubernetes.io/is-default-class=true
  Provisioner:           hyperflex.io/hxvolume
  Parameters:
  AllowVolumeExpansion:
  MountOptions:
  ReclaimPolicy:         Delete
  VolumeBindingMode:     Immediate
  Events:                \
```

Step 3 Create the persistent volume claim to request for storage.

```
$ cat <<EOF > pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pv-claim
spec:
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
EOF
```

Note The `storageClassName` field is optional.

```
$ kubectl create -f pvc.yaml
persistentvolumeclaim "pv-claim" created
```

Step 4 Verify if the persistent volume claim (pvc) is created.

```
$ kubectl describe pvc pv-claim
Name:          pv-claim
Namespace:     default
StorageClass:  standard
Status:        Bound
Volume:        hx-default-pv-claim-5c4e8978-cdd2-11e8-9a07-005056b8fd7b
Labels:
```

```
Annotations:   pv.kubernetes.io/bind-completed=yes
               pv.kubernetes.io/bound-by-controller=yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      3Gi
Access Modes:  RWO,ROX
Events:        \
```

Persistent Volume is automatically created and is bounded to this pvc.

Note When **VSPHERE** is used as the default storage class, a VMDK file is created inside the **kubevols** folder in the datastore which is specified during the creation of the tenant Kubernetes cluster.

Step 5 Create a pod that uses persistent volume claim with storage class.

```
$ cat <<EOF > pvc-pod.yaml
kind: Pod
apiVersion: v1
metadata:
  name: pvc-pod
spec:
  volumes:
    - name: pvc-storage
      persistentVolumeClaim:
       claimName: pv-claim
  containers:
    - name: pvc-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: pvc-storage
EOF

$ kubectl create -f pvc-pod.yaml
pod "pvc-pod" created
```

Step 6 Verify if the pod is up and running.

```
$ kubectl get pod pvc-pod

NAME      READY     STATUS    RESTARTS   AGE
pvc-pod   1/1       Running   0          16s
```

When **VSPHERE** is used as the default storage class, you can access vCenter and view the dynamically provisioned VMDKs of the pod.

Deploying Cafe Application with Ingress

This scenario describes deploying and configuring the *Cafe application* with Ingress rules to manage incoming HTTP requests. It uses a **Simple fanout with SSL termination Ingress**.

For more information on Ingress, see Load Balancing Kubernetes Services using NGINX.

Step 1 Go to the following URL:

https://github.com/nginxinc/kubernetes-ingress/tree/master/examples/complete-example

Step 2 Download the following yaml files:

- `tea-rc.yaml`
- `tea-svc.yaml`
- `coffee-rc.yaml`
- `coffee-svc.yaml`
- `cafe-secret.yaml`
- `cafe-ingress.yaml`

Step 3 Open the **kubectl** utility.

Step 4 Obtain the IP address of the L7 NGINX load balancer that Cisco Container Platform automatically installs:

```
$ kubectl get pods --all-namespaces -l app=ingress-nginx -o wide

NAMESPACE     NAME            READY  STATUS  RESTARTS AGE  IP            NODE
ingressnginx  nginx-           1/1   Running  0       3d   10.10.45.235  test-clusterwc5729f9ce2
              ingresscontroller
              -66974b775-jnmpl
```

Step 5 Deploy the Cafe application.

a) Create the coffee and the tea services and replication controllers:

```
kubectl create -f tea-rc.yaml<br>
kubectl create -f tea-svc.yaml<br>
kubectl create -f coffee-rc.yaml<br>
kubectl create -f coffee-svc.yaml
```

Step 6 Configure load balancing.

a) Create a Secret with an SSL certificate and a key:

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

b) Create an Ingress Resource:

```
kubectl create -f cafe-ingress.yaml
```

Step 7 Verify that the Cafe application is deployed.

```
$ kubectl get pods -o wide

NAMESPACE       READY STATUS     RESTARTS  AGE   IP                NODE
coffee-rc-jb9sx 1/1   Running    0         3d    192.168.151.134   test-cluster-wb3d42afeff
coffee-rc-tjwgj 1/1   Running    0         3d    192.168.44.133    test-cluster-wc5729f9ce2
tea-rc-6qmvm    1/1   Running    0         3d    192.168.44.132    test-cluster-wc5729f9ce2
tea-rc-ms46j    1/1   Running    0         3d    192.168.151.132   test-cluster-wb3d42afeff
tea-rc-tnftv    1/1   Running    0         3d    192.168.151.133   test-cluster-wb3d42afeff
```

Step 8 Verify if the coffee and tea services are deployed.

```
$ kubectl get svc

NAME         TYPE        CLUSTER-IP    EXTERNAL-IP     PORT(S)     AGE
coffee-svc   ClusterIP   10.105.139.1  80/TCP          3d
kubernetes   ClusterIP   10.96.0.1     443/TCP         4d
tea-svc      ClusterIP   10.109.34.129 80/TCP          3d
```

Step 9 Verify if the Ingress is deployed.

```
$ kubectl describe ing
```

```
Name: cafe-ingress
Namespace: default
Address:
Default backend: default-http-backend:80 (<none>)
TLS: cafe-secret terminates cafe.example.com
Rules:

Host               Path       Backends
cafe.example.com
                   /tea       tea-svc:80 (<none>)
                   /coffee    coffee-svc:80 (<none>)

Annotations:
Events: <none>
```

Step 10 Test the application.

a) Access the load balancer IP address `10.10.45.235`, which is obtained in Step2.

b) Test if the Ingress controller is load balancing as expected.

```
$ curl --resolve cafe.example.com:443:10.10.45.235 https://cafe.example.com/coffee --insecure
<!DOCTYPE html>
...
<p><span>Server address:</span> <span>192.168.151.134:80</span></p>
...
$ curl --resolve cafe.example.com:443:10.10.45.235 https://cafe.example.com/coffee --insecure
<!DOCTYPE html>
...
<p><span>Server address:</span> <span>192.168.44.133:80</span></p>
...
```