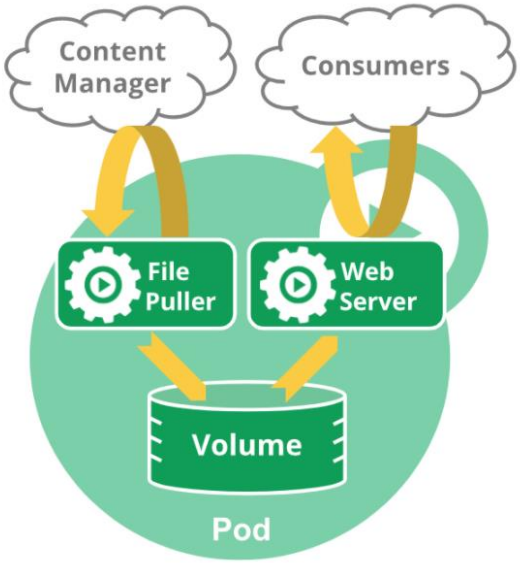
Kubernetes Basics

kubectl

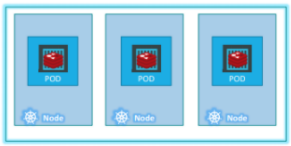
- kubectl is (almost) the only tool we'll need to talk to Kubernetes
- It is a rich CLI tool around the Kubernetes API
- Everything you can do with kubectl, you can do directly with the API
- kubectl can be pronounced "Cube C T L", "Cube cuttle", "Cube cuddle"...

PODs

- The application is developed and built into Docker Images and it is availalble on a Docker repository like Docker hub, so kubernetes can pull it down. We also assume that the Kubernetes cluster has already been setup and is working. This could be a single-node setup or a multi-node setup, doesn't matter. All the services need to be in a running state.
- The containers are encapsulated into a Kubernetes object known as PODs. A POD is a single instance of an application. A POD is the smallest object, that you can create in kubernetes.
- You have a single node kubernetes cluster with a single instance of your application running in a single docker container encapsulated in a POD.



- PODs usually have a one-to-one relationship with containers running your application. But if needed, you can encapsulate more than one container in a POD. These extra containers are generally helper containers. To scale UP you create new PODs and to scale down you delete PODs. You do not add additional containers to an existing POD to scale your application.



- Use the `kubectl get pods` command to see a list of pods available.

> To see detailed information about the pod run the `kubectl describe pod` command. This will tell you information about the POD, when it was created, what labels are assigned to it, what docker containers are part of it and the events associated with that POD.

```
kubectl run nginx --image nginx
```

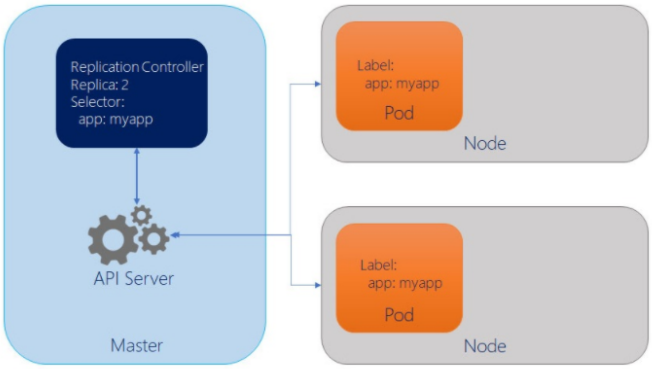
```
kubectl get pods

NAME                     READY   STATUS              RESTARTS   AGE
nginx-76df748b9-hjnc2    0/1     ContainerCreating   0          18s
```

```
kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-76df748b9-hjnc2    1/1     Running   0          20m
```

Replication Controller

> A ReplicationController ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.



- To prevent users from losing access to our application, we would like to have more than one instance or POD running at the same time. That way if one fails we still have our application running on the other one. The replication controller helps us run multiple instances of a single POD in the kubernetes cluster thus providing High Availability
- The replication controller can help by automatically bringing up a new POD when the existing one fails. Thus the replication controller ensures that the specified number of PODs are running at all times.
- If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, you should use a ReplicationController even if your application requires only a single pod. A ReplicationController is similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.
- ReplicationController is often abbreviated to "rc".

Running an example ReplicationController

1. **controllers/replication.yaml**

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Tips:

- **kind: ReplicationController** We have defined the kind as replication controller which tells the kubectl that the yaml file is going to be used for creating the replication controller.
- **name: nginx** This helps in identifying the name with which the replication controller will be created. If we run the kubctl, get rc nginx it will show the replication controller details.
- **replicas: 3** This helps the replication controller to understand that it needs to maintain three replicas of a pod at any point of time in the pod lifecycle.
- **name: nginx** In the spec section, we have defined the name as nginx which will tell the replication controller that the container present inside the pods is nginx.
- **containerPort: 80** It helps in making sure that all the nodes in the cluster where the pod is running the container inside the pod will be exposed on the same port 7474.

2. Run the example job by downloading the example file and then running this command:

```
kubectl apply -f https://k8s.io/examples/controllers/replication.yaml
```

3. Check on the status of the ReplicationController using this command:

```
kubectl describe replicationcontrollers/nginx
```

Note: During the evaluation of k8s, in practice, ReplicationController is replaced by Deployments and not preferred in the community any more. Deployments are automatically create the replication sets.

Look at the sample output:

```
ubuntu@master:~$ kubectl get deployments
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           12m
```

`nginx-deployment` is the deployment name.

```
ubuntu@master:~$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-6b474476c4   3         3         3       13m
```

`nginx-deployment-6b474476c4` is the replication set name. The k8s created replication set by default and named it by adding the `6b474476c4` number to the deployment name.

```
ubuntu@master:~$ kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-6b474476c4-55klw   1/1     Running   0          18m
nginx-deployment-6b474476c4-gjzvr   1/1     Running   0          18m
nginx-deployment-6b474476c4-rg4df   1/1     Running   0          18m
```

Note the pod numbers added to the `nginx-deployment-6b474476c4` replication set name.

Replication Sets

> A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

- A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.

Deployment

> Deployment is a method of converting images to containers and then allocating those images to pods in the Kubernetes cluster. A Deployment provides declarative updates for Pods and ReplicaSets.

- You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Creating a Deployment

1. The following is an example of a Deployment. It creates a ReplicaSet to bring up three nginx Pods:

```
controllers/nginx-deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

2. Run `kubectl get deployments` to check if the Deployment was created.
3. To see the Deployment rollout status, run `kubectl rollout status deployments/nginx-deployment`.
4. Run the `kubectl get deployments` again a few seconds later.
5. To see the ReplicaSet (rs) created by the Deployment, run `kubectl get rs`.
6. To see the labels automatically generated for each Pod, run `kubectl get pods --show-labels`.

Volumes

> Storage is managed by the storage/system administrators. The end user will just receive instructions to use the storage but is not involved with the underlying storage management. A volume can be thought of as a directory which is accessible to the containers in a pod.

- As soon as the life of a pod ended, the volume was also lost.
- On the other hand, the volumes that are created through Kubernetes is not limited to any container. It supports any or all the containers deployed inside the pod of Kubernetes. A key advantage of Kubernetes volume is, it supports different kind of storage wherein the pod can use multiple of them at the same time.
- A directory which is mounted inside a Pod is backed by the underlying Volume Type. Examples of the Volume Types that support managing storage using PersistentVolumes are:
- emptyDir : An empty Volume is created for the Pod as soon as it is scheduled on the worker node. (Node-local memory)
- hostPath : With the hostPath Volume Type, we can share a directory from the host to the Pod. If the Pod is terminated, the content of the Volume is still available on the host. (Node-local memory)
- awsElasticBlockStore Cloud volumes
- azureDisk Cloud volumes
- gcePersistentDisk (Google Compute Engine (GCE) persistent disk) Cloud volumes
- nfs
- persistentVolumeClaim (Special volume type)

With cloud volumes, nfs, and PersistentVolumeClaim, the volume is independent and placed outside of the pod. Persistent Volume Claims are the core solution for persistent volumes in Kubernetes. We will cover this persistentVolumeClaim.

PersistentVolumesClaims

- Persistent volumes life-cycle is independent from any pod using it. Thus, persistent volumes are perfect for use cases in which you need to retain data regardless of the unpredictable life process of Kubernetes pods.
- Without persistent volumes, maintaining services as common as a database would be impossible. Whenever a pod gets replaced, the data gained during the life-cycle of that pod would be lost. However, thanks to persistent volumes, data is contained in a consistent state.
- A PersistentVolumeClaim (PVC) is a request for storage by a user. Users request for PersistentVolume resources based on type, access mode, and size. There are three access modes: ReadWriteOnce (read-write by a single node), ReadOnlyMany (read-only by many nodes), and ReadWriteMany (read-write by many nodes). Once a suitable PersistentVolume is found, it is bound to a PersistentVolumeClaim.

How to Create a Persistent Volume

1. To create the following clarusway-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

2. Then, deploy the persistent volume by using the following command with the name of the .yaml file you created:

```
kubectl create -f clarusway-pv.yaml
```

3. You can view it by running:

```
kubectl get pv
```

You can verify with a simple tutorial that uses this file from this link

Service

> Kubernetes Services enable communication between various components within and outside of the application. Kubernetes Services helps us connect applications together with other applications or users.

- A service is a stable endpoint to connect to "something"
- List the services on our cluster with one of these commands:

```
kubectl get services

NAME           TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP      10.96.0.1               443/TCP        35m
my-nginx-svc   LoadBalancer   10.109.139.12        80:31462/TCP   20m
```

or

```
kubectl get svc

NAME           TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP      10.96.0.1               443/TCP        33m
my-nginx-svc   LoadBalancer   10.109.139.12        80:31462/TCP   18m
```

- **NodePort :** It will expose the service on a static port on the deployed node. A ClusterIP service, to which NodePort service will route, is automatically created. The service can be accessed from outside the cluster using the NodeIP:nodePort.
- **ClusterIP :** This helps in restricting the service within the cluster. It exposes the service within the defined Kubernetes cluster.

```
curl -k https://10.96.0.1
```

> -k is used to skip certificate verification
>
> Make sure to replace 10.96.0.1 with the CLUSTER-IP shown by kubectl get svc

- **LoadBalancer :** It uses cloud providers' load balancer. NodePort and ClusterIP services are created automatically to which the external load balancer will route.

> A NodePort service is created, and the load balancer sends traffic to that port