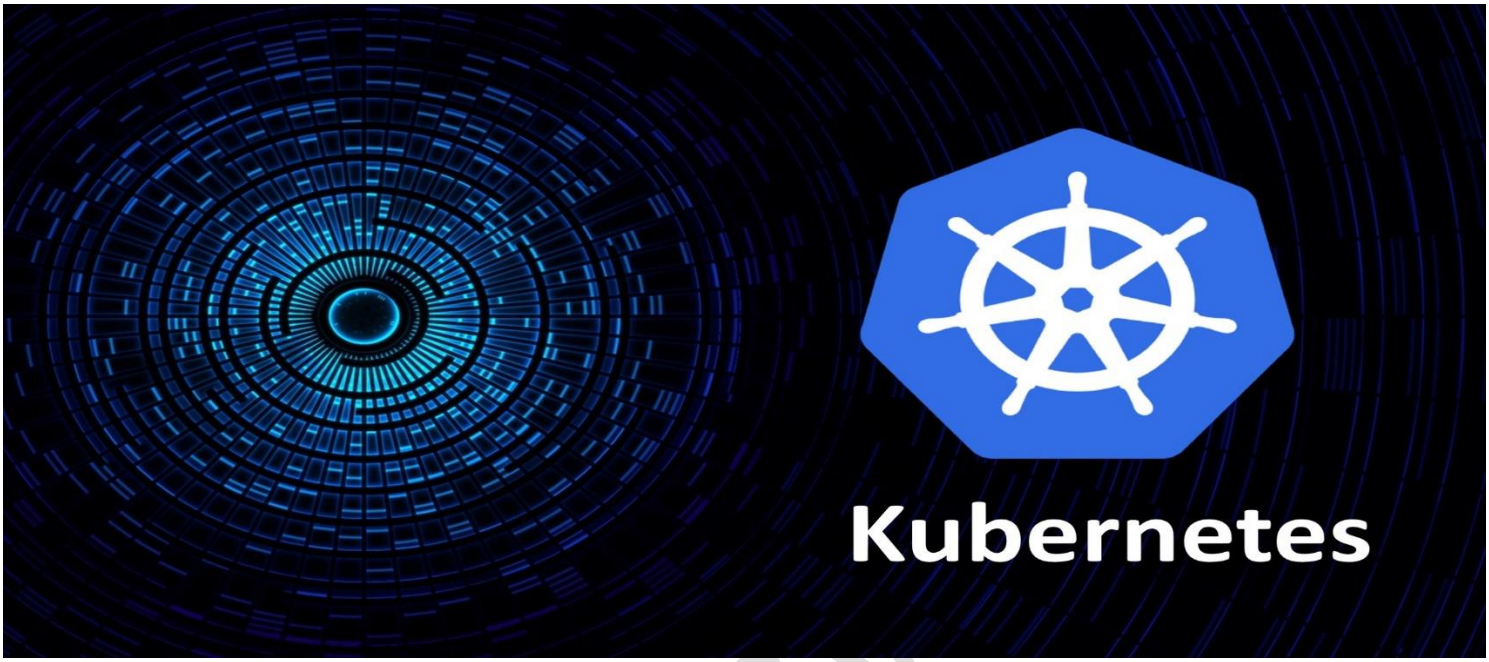


Software Deployments

Software deployment includes all the process required for preparing a software application to run and operate in a specific environment.

It involves installation, configuration, testing and making changes to optimize the performance of the software.

Types of Deployments?

- ✓ **Recreate: Version A is terminated then version B is rolled out.**
- ✓ **Ramped (also known as rolling-update or incremental): Version B is slowly rolled out and replacing version A.**
- ✓ **Blue/Green: Version B is released alongside version A, then the traffic is switched to version B.**
- ✓ **Canary: Version B is released to a subset of users, then proceed to a full rollout.**

Streamline Deployments using Docker?

- Compose the application into Docker Images
- Break the application components into individual containers
- Split the data that's shared between services into volumes
- Separate responsibilities so that each containers runs only one component/executable
- Store the changeable data (configurations, logs) as Volumes so that they are mounted on various containers

Modern development's primary focus is often based around three central concepts:

- ✓ efficiency
- ✓ reliability
- ✓ repeatability

Why do we want a Container Orchestration System?

Imagine that you had to run hundreds of containers. You will need from a management angle to make sure that the cluster is up and running and have necessary features like:

- Health Checks on the Containers
- Launching a fixed set of Containers for a particular Docker image
- Scaling the number of Containers up and down depending on the load
- Performing rolling update of software across containers

What is Kubernetes?

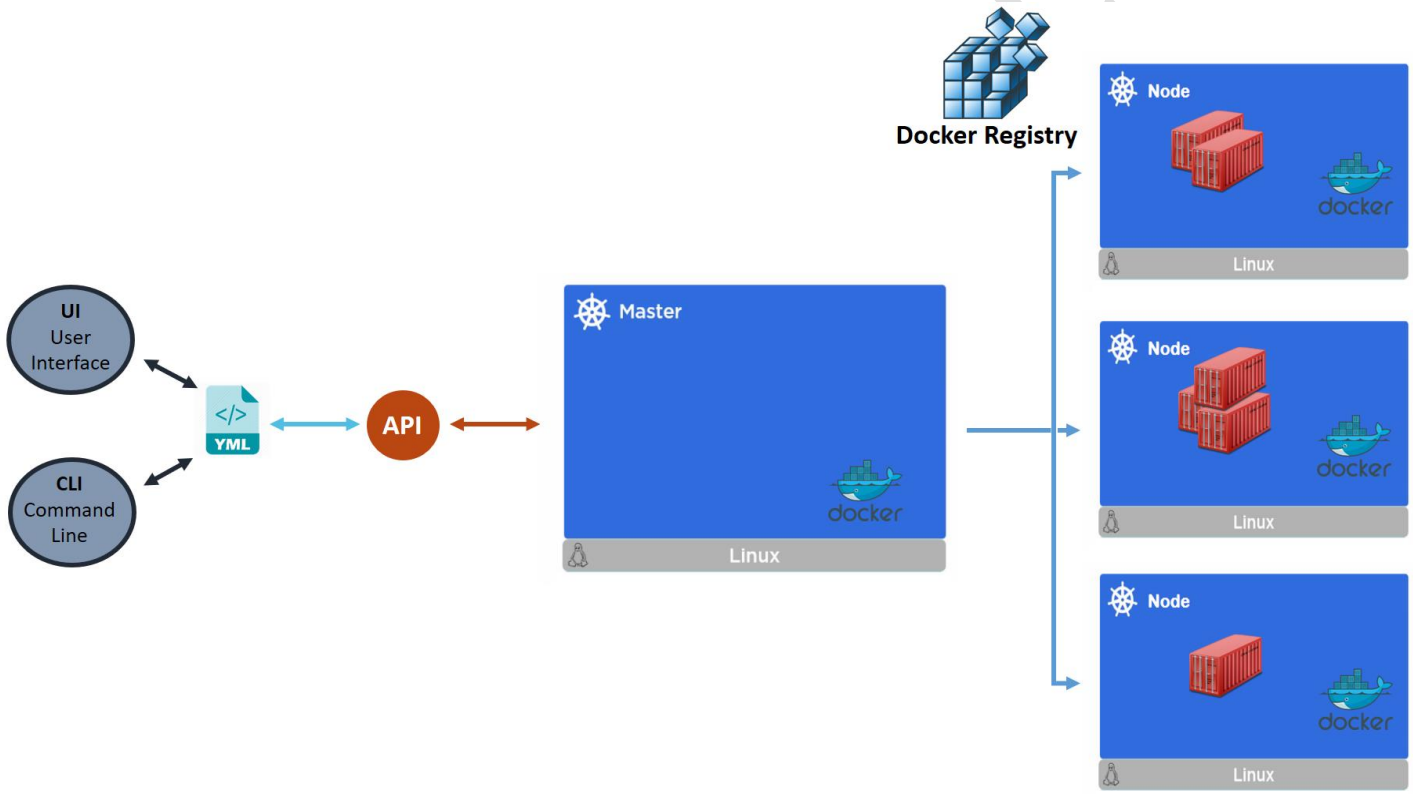
- K8s is an open-source container-orchestration system for automating deployment, scaling and management of containerized applications across clusters of hosts.
- At a high level, it takes multiple Containers running on different hosts and lets you use them together.
- Run containers anywhere on clusters of physical, virtual machines and cloud infrastructure.
- Start, stop, update, and manage a cluster of machines running containers in a consistent and maintainable way.
- A declarative language for launching containers.
- We will define the desired state, K8s will monitor the current actual state and synchronize it with the desired state.
- K8s provides the tooling to manage the when, where, and how many of the application stack and its components.
- K8s also allows finer control of resource usage, such as CPU, memory, and disk space across our infrastructure.

Features of Kubernetes

Following are some of the important features of Kubernetes:

- Continuous development, integration and deployment
- Containerized infrastructure
- Application-centric management
- Auto-scalable infrastructure
- Environment consistency across development testing and production
- Loosely coupled infrastructure, where each component can act as a separate unit
- Higher density of resource utilization

Kubernetes Architecture



Infrastructure for Kubernetes Installation

- Localhost Installation - Minikube
- On-Premise Installation - Vagrant, Vmware, KVM
- Cloud Installation
  - ✓ Google Kubernetes Engine (GKE)
  - ✓ Azure Container Service (AKS)
  - ✓ Amazon Elastic Container Service for Kubernetes (EKS)
  - ✓ OpenShift
  - ✓ Platform9
  - ✓ IBM Cloud Container Service

Kubernetes Installation Tools/Resources:

- ❑ Kubeadm
- ❑ Kubespray
- ❑ kops

Setting Up a Single-Node Kubernetes Cluster with Minikube (EC2 Ubuntu)

✓ Install Docker

```
$ sudo apt update && apt -y install docker.io
```

✓ Install kubectl

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl && chmod +x ./kubectl && sudo mv ./kubectl /usr/local/bin/kubectl
```

✓ Install Minikube

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

- Start Minikube

```
$ minikube start --vm-driver=none
$ minikube status
```

Kubectl Usage

$ kubectl [COMMAND] [TYPE] [NAME] [flags]

COMMAND: Specifies the operation that you want to perform on one or more resources, for example create, apply, get, describe, delete, exec …

TYPE: Specifies the resource type i.e nodes, pods, services, rc, rs …

NAME: Specifies the name of the resource

flags: Optional arguments

$ kubectl version

$ kubectl get node

$ kubectl describe nodes <nodename>

Kubernetes Objects

- Kubernetes uses Objects to represent the state of your cluster
  - What containerized applications are running (and on which nodes)
  - The resources available to those applications
  - The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance
- Once you create the object, the Kubernetes system will constantly work to ensure that object exists and maintain cluster's desired state
- Every Kubernetes object includes two nested fields that govern the object's configuration: the object spec and the object status
- The spec, which we provide, describes your desired state for the object-the characteristics that you want the object to have.
- The status describes the actual state of the object, and is supplied and updated by the Kubernetes system.
- All objects in the kubernetes are identified by a Unique Name and a UID.

The basic Kubernetes objects include:

- ❑ Pod
- ❑ Service
- ❑ Volume
- ❑ Namespace

In addition, Kubernetes contains a number of higher-level abstractions called Controllers. Controllers build upon the basic objects, and provide additional functionality:

- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet

Kubernetes Objects Management

- The kubectl command-line tool supports several different ways to create and manage Kubernetes objects

Management technique	Operates on	Recommended environment
Imperative commands	Live objects	Development projects
Declarative object configuration	Individual files ((yaml/json)	Production

- Declarative is about describing what you're trying to achieve, without instructing how to do it
- Imperative explicitly tells "how" to accomplish it

YAML Basics

- Kubernetes uses YAML syntax for expressing object configuration playbooks
- Nearly every YAML file starts with a list
- Each item in the list is a list of key/value pairs, commonly called a "hash" or a "dictionary"
- All YAML files can optionally begin with "---" and end with "..."
- All members of a list are lines beginning at the same indentation level starting with a "- "

```
--- # A list of tasty fruits
fruits:
  - Apple
  - Orange
  - Strawberry
```

```
  - Mango
```

- A dictionary is represented in a simple key: value form (the colon must be followed by a space)

```
--- # An employee record
Employee:
  name: ADAM
  job: DevOps Engineer
    skill: Elite
```

Pods

A Pod is the smallest and simplest unit in the Kubernetes object model that you create or deploy.

- A Pod represents a running process on your cluster.
- A Pod encapsulates an application container, storage resources, a unique network IP, and options that govern how the container(s) should run
- Kubernetes manages the Pods rather than the containers directly.
- Pods in a Kubernetes cluster can be used in two main ways:
- Pods that run a single container - common usecase
- Pods that run multiple containers that need to work together
- Each Pod is assigned a unique IP address.
- Every container in a Pod shares the IP address and network ports.
- Containers inside a Pod can communicate with one another using localhost.
- All containers in the Pod can access the shared volumes, allowing those containers to share data.
- Kubernetes scales pods and not containers.

Understanding Pods

- When a Pod gets created, it is scheduled to run on a Node in your cluster.
- The Pod remains on that Node until the process is terminated, the pod object is deleted, the pod is evicted for lack of resources, or the Node fails.
- If a Pod is scheduled to a Node that fails, or if the scheduling operation itself fails, the Pod is deleted
- If a node dies, the pods scheduled to that node are scheduled for deletion, after a timeout period.

- A given pod (UID) is not "rescheduled" to a new node; instead, it will be replaced by an identical pod, with even the same name if desired, but with a new UID
- Volumes in a pod will exists as long as that pod (with that UID) exists. If that pod is deleted for any reason, volume is also destroyed and created as new on new pod
- Kubernetes uses a Controller, that handles the work of managing the Pod instances.
- A Controller can create and manage multiple Pods, handling replication, rollout and providing self-healing capabilities

YML template for POD

- Always means that the container will be restarted even if it exited with a zero exit code
- OnFailure means that the container will only be restarted if it exited with a non-zero exit code
- Never means that the container will not be restarted regardless of why it exited.

Syntax:

```
kind: Pod                       # Object Type
apiVersion: v1                  # API version
metadata:                       # Set of data which describes the Object
  name: <Pod Name>              # Name of the Object
spec:                           # Data which describes the state of the Object
  containers:                   # Data which describes the Container details
    - name: <ContainerName1>                # Name of the Container1
      image: <ImageName1>               # Base Image which is used to create Container1
      command: [<Comma separated first cmd>]
    - name: <ContainerName2>                # Name of the Container2
      image: <ImageName2>               # Base Image which is used to create Container2
      command: [<Comma separated first cmd>]
  restartPolicy: Never          # Defaults to Always
```

Example: Check class notes for example

- Create or update an Object
  $ kubectl create -f pod1.yml (or)
  $ kubectl apply -f pod1.yml

- Get the list of the pod objects available
  $ kubectl get pods
  $ kubectl get pods -o wide

```
$ kubectl get pods --all-namespaces
```

- Get the details of the pod object
  ```
  $ kubectl describe pod <podname>
  ```
- Get running logs from the container inside the pod object
  ```
  $ kubectl logs -f <podname>
  $ kubectl logs -f <podname> -c <containername>
  ```

- Run OS commands in an existing pod
  ```
  $ kubectl exec <podname> -- <OScmd>
  ```

- Run OS commands in an existing container (multi container pod)
  ```
  $ kubectl exec <podname>  -c <containername> -- <OScmd>
  ```

- Attach to the running container interactively
  ```
  $ kubectl exec <podname> -i -t -- /bin/bash (or)
  $ kubectl attach <podname> -i
  ```

- Delete a Pod
  ```
  $ kubectl delete pods <podname> (or)
  $ kubectl delete -f <YAML>
  ```

Pod Environment Variables

Syntax:

```
kind: Pod
apiVersion: v1
metadata:
  name: <PodName>
spec:
  containers:
    - name: <ContainerName>
      image: <ImageName>
      command: [<first cmd>]
      env:          # List of environment variables to be used inside the pod
      - name: <Name of the environment variable>
        value: <value>
```

Example: Check class notes for example

$ kubectl apply -f podmulcont.xml

$ kubectl exec environments - env

Managing Compute Resources for Containers

- A pod in Kubernetes will run with no limits on CPU and memory
- You can optionally specify how much CPU and memory (RAM) each Container needs.
- Scheduler decides about which nodes to place Pods, only if the Node has enough CPU resources available to satisfy the Pod CPU request
- CPU is specified in units of cores, and memory is specified in units of bytes.
- Two types of constraints can be set for each resource type: requests and limits
  - ✓ A request is the amount of that resources that the system will guarantee for the container & Kubernetes will use this value to decide on which node to place the pod
  - ✓ A limit is the maximum amount of resources that Kubernetes will allow the container to use. In the case that request is not set for a container, it defaults to limit. If limit is not set, then if defaults to 0 (unbounded).
- CPU values are specified in “millicpu” and memory in MiB

Syntax:

```
apiVersion: v1
kind: Pod
metadata:
  name: <PodName>
spec:
  containers:
  - name: <ContainerName>
    image: <ImageName>
    command: [<first cmd>]
    resources:                          # Describes the type of resources to be used
      requests:
        memory: "<value in mebibytes>" # A mebibyte is 1,048,576 bytes, ex: 64Mi
        cpu: "<value in millicores >"        # CPU core split into 1000 units (milli = 1000), ex: 100m
      limits:
```

memory: "<value in mebibytes>" # ex: 128Mi

cpu: "<value in millicores >" # ex: 200m

Example: Check class notes for example

Labels & Selectors

- Labels are the mechanism you use to organize Kubernetes objects
- A label is a key-value pair without any predefined meaning that can be attached to the Objects
- Labels are similar to Tags in AWS or Git where you use a name to quick reference
- So you're free to choose labels as you need it to refer an environment which is used for Dev or Testing or Production, refer an product group like DepartmentX, DepartmentY
- Multiple labels can be added to a single object

Syntax:

```
kind: Pod
apiVersion: v1
metadata:
  name: <PodName>
  labels:                                  # Specifies the Label details under it
    <Keyname1>: <value>
    <Keyname2>: <value>
spec:
    containers:
      - name: c00
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

- Apply the yaml

$ kubectl apply -f <yaml>

- To see list of pods available with details of Labels if any attached to it

$ kubectl get pods --show-labels

- Add a label to an existing pod

$ kubectl label pods <podname> <labelkey>=<value>

- List pods matching a label

$ kubectl get pods -l <label>=<value>

- We can also delete pods based on label selection

$ kubectl delete pods -l <label>=<value>

Label Selectors:

- Unlike name/UIDs, Labels do not provide uniqueness, as in general, we can expect many objects to carry the same label
- Once labels are attached to an object, we would need filters to narrow down and these are called as Label Selectors
- The API currently supports two types of selectors: *equality-based* and *set-based*
- A label selector can be made of multiple *requirements* which are comma-separated

Equality-based requirement: (= , !=)

environment = production

tier != frontend

Set-based requirement: (in,notin and exists)

environment in (production, qa)

tier notin (frontend, backend)

- K8s also support set-based selectors i.e match multiple values

$ kubectl get pods -l 'label in (value1, value2)'

- List pods matching multiple values

$ kubectl get pods -l environment=testing, tier=frontend

Node Selectors:

- One use case for selecting labels is to constrain the set of nodes onto which a pod can schedule i.e You can tell a pod to only be able to run on particular nodes
- Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement, but on certain circumstances we might need it
- We can use labels to tag Nodes
- You the nodes are tagged, you can use the Label Selectors to specify the pods run only of specific nodes

- First we tag the node
- Use nodeSelector to the pod configuration

Syntax:

```
kind: Pod
apiVersion: v1
metadata:
  name: <PodName>
  labels:
    <Keyname>: <value>
spec:
    containers:
      - name: c00
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
    nodeSelector:                    # specifies which node to run the pod
      <Keyname>: <value>
```

- Apply the yaml

```
$ kubectl apply -f <yaml>
```

- You would see the status for this new pod is pending, as there are no nodes which has the label

```
$ kubectl get pods
$ kubectl describe pods <podname>
```

- Add a label to an node

```
$ kubectl label nodes <nodename> <labelkey>=<value>
```

- Describe the Node & Pod

```
$ kubectl describe node <nodename>
$ kubectl get pods
```

Scaling & Replication

- Kubernetes was designed to orchestrate multiple containers and replication.
- Need for multiple containers/replication helps us with these:
- Reliability: By having multiple versions of an application, you prevent problems if one or more fails.
- Load balancing: Having multiple versions of a container enables you to easily send traffic to different instances to prevent overloading of a single instance or node
- Scaling: When load does become too much for the number of existing instances, Kubernetes enables you to easily scale up your application, adding additional instances as needed
- Rolling updates: updates to a service by replacing pods one-by-one

Replication Controller

- A Replication Controller is a object that enables you to easily create multiple pods, then make sure that that number of pods always exists.
- If a pod created using RC will be automatically replaced if they does crash,fail, deleted or terminated
- Using RC is recommended if you just want to make sure 1 pod is always running, even after system reboots
- You can run the RC with 1 replica & the RC will make sure the pod is always running

Syntax:

```
kind: ReplicationController          # this defines to create the object of replication type
apiVersion: v1
metadata:
  name: <Name Of RC>
spec:
  replicas: <Number>                 # this element defines the desired number of pods
  selector:          # tells the controller which pods to watch/belong to this Replication Controller
    <Keyname>: <value>                          # these must match the labels
  template:                          # template element defines a template to launch a new pod
    metadata:
      name: <PodName
      labels:        # selector values need to match the labels values specified in the pod template
        <Keyname>: <value>
    spec:
      containers:
```

```
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

- Apply the yaml

```
$ kubectl apply -f <yaml>
$ kubectl get pods --show-labels
$ kubectl get pods -l keyname=value
$ kubectl get rc
```

- Delete the pod & RC will recreate it

```
$ kubectl delete pods <podname>
$ kubectl describe rc <replicationcontrollername>
```

- Scale Replicas:

```
$ kubectl scale --replicas=<num>  rc/<replicationcontrollername>    (OR)
$ kubectl scale --replicas=<num> <resourcetype> -l <key>=<value>
```

- Delete Replicationcontroller:

```
$ kubectl delete rc <replicationcontrollername>
```

Replication Set

- ReplicaSet is the next generation Replication Controller
- The replication controller only supports equality-based selector whereas the replica set supports set-based selector i.e filtering according to set of values
- ReplicaSet rather than the Replication Controller is used by other objects like Deployment

Syntax:

```
kind: ReplicaSet                    # Defines the object to be ReplicaSet
apiVersion: apps/v1                 # Replicaset is not available on v1
metadata:
  name: <RS Name>
spec:
  replicas: 2                       # this element defines the desired number of pods
```

```
 selector:              # tells the controller which pods to watch/belong to this Replication Set
   matchExpressions:                    # these must match the labels
    - {key: keyname, operator: In, values: [value1, value2, value3]}
    - {key: keyname, operator: NotIn, values: [value1]}
 template:                  # template element defines a template to launch a new pod
   metadata:
    name: <podname>
    labels:          # selector values need to match the labels values specified in the pod template
      <keyname>: <value>
   spec:
    containers:
     - name: c00
       image: ubuntu
       command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

- Apply Replicationcontroller:

```
$ kubectl apply -f rs.yml
$ kubectl get rs
$ kubectl describe rs <replicasetname>
```

- Scale Replicas:

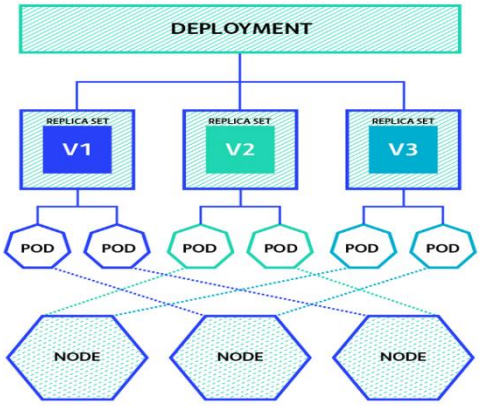
```
$ kubectl scale --replicas=1  rs/<replicasetname> (OR)
$ kubectl scale --replicas=2 rs -l keyname=value
```

- Delete ReplicaSet:

```
$ kubectl delete rs <replicasetname>
```

Deployments

- Just using RC & RS might be cumbersome to deploy apps, update or rollback apps in the cluster
- A Deployment object acts as a supervisor for pods, giving you fine-grained control over how and when a new pod is rolled out, updated or rolled back to a previous state
- When using Deployment object, we first define the state of the App, then k8s master schedules mentioned app instance onto specific individual Nodes
- K8s then monitors, if the Node hosting an instance goes down or pod is deleted, the Deployment controller replaces it.
- This provides a self-healing mechanism to address machine failure or maintenance.



The following are typical use cases for Deployments:

- Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
- Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
- Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- Scale up the Deployment to facilitate more load.
- Pause the Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- Use the status of the Deployment as an indicator that a rollout has stuck.
- Clean up older ReplicaSets that you don't need anymore

Syntax:

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: <DeploymentName>
spec:
  replicas: <Number of Containers>
  template:
    metadata:
      name: <PodName>
      labels:
        <Keyname>: <Value>
    spec:
     containers:
       - name: c00
         image: ubuntu
         command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

- Creating a Deployment

```
$ kubectl apply -f pod8.xml
$ kubectl get deploy
$ kubectl describe deploy/<deploymentname>
$ kubectl get rs
$ kubectl get pods
```

- Updating a Deployment (Change the echo cmd & save the yml content as new file & apply)

```
$ kubectl apply -f <yml>
$ kubectl get pods
```

✓ We will see the rollout of two new pods with the updated cmd, as well as old pods being terminated

- ✓ Also, a new replica set has been created by the deployment

$ kubectl get rs

- Recovering from crashes

$ kubectl delete pod <podname>

- Scaling a Deployment

$ kubectl scale --replicas=<num> deployment/<deploymentname>

Manage the rollout of a resource

Usage $ kubectl rollout SUBCOMMAND OBJECT

history	View rollout history
pause	Mark the provided resource as paused
resume	Resume a paused resource
status	Show the status of the rollout
undo	Undo a previous rollout

- Deployment status

$ kubectl rollout status deployment/<deploymentname>

- Deployment history

$ kubectl rollout history deploy/<deploymentname>

- Rolling Back a Deployment

If there are problems in the deployment Kubernetes will automatically roll back to the previous version, however you can also explicitly roll back to a specific revision, as in our case to revision 1 (the original pod version)

$ kubectl rollout undo deploy/<deploymentname> --to-revision=1

$ kubectl rollout undo deploy/<deploymentname>

- Set the image of the deployment

$ kubectl set image deployment/<deploymentname> <containername>=<image>

- Remove the deployment, with it the replica sets and pods it supervises

$ kubectl delete deploy <deploymentname>

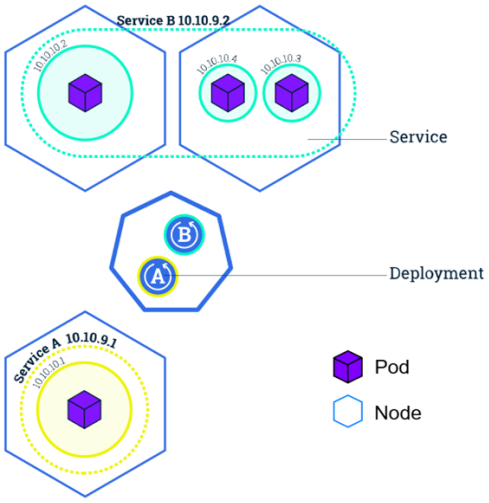
Pausing and Resuming a Deployment

$ kubectl rollout pause deploy/<deploymentname>

$ kubectl rollout resume deploy/<deploymentname>

Services

- When using RC, pods are terminated and created during scaling or replication operations
- When using Deployments, while updating the image version the pods are terminated & new pods take the place of older pods.
- Pods are very dynamic i.e they come & go on the k8s cluster and on any of the available nodes & it would be difficult to access the pods as the pods IP changes once its recreated
- Service Object is an logical bridge between pods & endusers, which provides Virtual IP (VIP) address.
- Service allows clients to reliably connect to the containers running in the pod using the VIP.
- The VIP is not an actual IP connected to a network interface, but its purpose is purely to forward traffic to one or more pods.
- kube-proxy is the one which Keeps the mapping between the VIP and the pods up-to-date, which queries the API server to learn about new services in the cluster.
- Although each Pod has a unique IP address, those IPs are not exposed outside the cluster.
- Services helps to expose the VIP mapped to the pods & allows applications to receive traffic
- Labels are used to select which are the Pods to be put under a Service.
- Creating a Service will create an endpoint to access the pods/Application in it.
- Services can be exposed in different ways by specifying a type in the Service Spec:
    - ClusterIP (default) - Exposes VIP only reachable from within the cluster.
    - NodePort - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>.
    - LoadBalancer - Created by cloud providers that will route external traffic to every node on the NodePort (ex: ELB on AWS).
- By default service can run only between ports 30000-32767



Pod with Ports

Syntax:

```
kind: Pod
apiVersion: v1
metadata:
  name: <NameofPod>
  labels:
    <keyname>: <value>                    # Label for applying service later
spec:
  containers:
    - name: <ContainerName>
      image: <ImageName>
      ports:
       - containerPort: <PortNumber>              # Expose port from container
```

Example: Check class notes for example

Service Syntax:

```
kind: Service                  # Defines to create Service type Object
apiVersion: v1
metadata:
  name: <Servicename>
spec:
  ports:
    - port: <PortNumber>                # Containers port exposed
      targetPort: <PortNumber>          # Pods port
  selector:
    <Key>: <value>             # Apply this service to any pods which has the specific label
  type: <type>                 # Specifies the service type i.e ClusterIP or NodePort
```

- List the services available in the cluster

$ kubectl apply -f service.yml

$ kubectl get svc

$ kubectl describe pod <podname>

- curl <podIP>:<PortNumber>

$ kubectl describe svc <servicename>

- curl <serviceIP>:< PortNumber>

- Delete a service

$ kubectl delete svc <servicename>

- Change the Service type to NodePort & try accessing the port 80 from Node

$ kubectl describe svc <servicename>

- http://<publicIP>:<NodePort-exposedby-service>

HealthChecks

- A Pod is considered ready when all of its Containers are ready.
- In order to verify if a container in a pod is healthy and ready to serve traffic, Kubernetes provides for a range of health checking mechanisms
- Health checks, or probes are carried out by the kubelet to determine when to restart a container (for livenessProbe) and used by services and deployments to determine if a pod should receive traffic (for readinessProbe).
- For example, 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.
- One use of readiness probes is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.
- For running healthchecks we would use cmds specific to the application.
- 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.

LivenessProbe with CMD Syntax:

apiVersion: v1

kind: Pod

metadata:

labels:

```
    <Key>: <Value>
  name: <Livenessname>
spec:
  containers:
  - name: <ContainerName>
    image: <ImageName>
    livenessProbe:                          # define the health check
      exec:
        command:                            # command to run periodically
        - <OS Cmd>
      initialDelaySeconds: <Seconds>      # Wait for the specified time before it runs the first probe
      periodSeconds: <Seconds>                    # Run the above command every n sec
      timeoutSeconds: <Seconds>          # Seconds to timeout if the cmd is not responding
```

LivenessProbe with URL Syntax:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    <Key>: <Value>
name: <PodName>
spec:
  containers:
  - name: <ContainerName>
    image: <ImageName>
      ports:
       - containerPort: <port>
      livenessProbe:
       initialDelaySeconds: <Seconds>
```

```
periodSeconds: <Seconds>
httpGet:                    # HTTP URL to check periodically
 path: /                    # Endpoint to check inside the container / means http://localhost/
 port: <Portnumber>                  # Specific port to check
```

Readiness check

- While both liveness and readiness check and describe the state of the service, they serve quite a different purpose.
- Liveness describes if the pod has started and everything inside it is ready to take the load. If not, then it gets restarted. This does not include any external dependencies like for example databases, which are clearly something the service doesn't have control over. We don't get much benefit if we restart the service when the database is down, because it will not help.
- This is where Readiness checks come in. This is basically a check if the application can handle the incoming requests.
- It should do a sanity check against both internal and external elements that make the service go, so the connection to the database should be checked there.
- If the check fails, the service is not being restarted, but the Kubernetes will not direct any traffic to it. This is particularly useful if we perform a rolling update and the new version has some issue connecting to the DB. In this case, it stays not ready, but those instances are not being restarted.

ReadinessProbe With URL Syntax

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    <Key>: <Value>
name: <PodName>
spec:
  containers:
  - name: <ContainerName>
    image: <ImageName>
      ports:
       - containerPort: <port>
```

```
readinessProbe:
 initialDelaySeconds: <Seconds>
 periodSeconds: <Seconds>
 httpGet:                    # HTTP URL to check periodically
  path: /                    # Endpoint to check inside the container / means http://localhost/
  port: <Portnumber>                    # Specific port to check
```

When to use Readiness and Liveness Probes

- Despite how great readiness and liveness probes can be, they're not always necessary. When updating deployments, Kubernetes will already wait for a replacement pod to start running before removing the old pod. Additionally, if a pod stops running, it will automatically try to restart it. Where these probes prove their worth is the time between when a pod starts running and when your service actually starts functioning. Kubernetes already knows if your container is running, probes let it know if your container is functioning.
- If one of our services take time between when a pod starts running and the service is actually responding to requests can be significant (5-10 seconds). Without a readiness probe, we'd end up with at least that much downtime every time we updated that deployment.
- Additionally, we have a number of services that can fail in ways that don't always result in the container crashing. These aren't particularly common, but when it happens, it's nice to have a livenessprobe around to catch the issue and restart the container.

Example: Check class notes for example

Volumes

- Containers are ephemeral (short lived in nature).
- All data stored inside a container is deleted if the container crashes. However, the kubelet will restart it with a clean state, which means that it will not have any of the old data
- To overcome this problem, Kubernetes uses Volumes. A Volume is essentially a directory backed by a storage medium. The storage medium and its content are determined by the Volume Type.
- In Kubernetes, a Volume is attached to a Pod and shared among the containers of that Pod.
- The Volume has the same life span as the Pod, and it outlives the containers of the Pod - this allows data to be preserved across container restarts
- A Volume Type decides the properties of the directory, like size, content, etc. Some examples of Volume Types are:

- node-local types such as emptyDir and hostPath
- file-sharing types such as nfs
- cloud provider-specific types like awsElasticBlockStore, azureDisk, or gcePersistentDisk
- distributed file system types, for example glusterfs or cephfs
- special-purpose types like secret, gitRepo
- persistentVolumeClaim

EmptyDir

- Use this when we want to share contents between multiple containers on the same pod & not to the host machine
- An emptyDir volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node.
- As the name says, it is initially empty.
- Containers in the Pod can all read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each Container.
- When a Pod is removed from a node for any reason, the data in the emptyDir is deleted forever.
- A Container crashing does not remove a Pod from a node, so the data in an emptyDir volume is safe across Container crashes.

Syntax:

```
apiVersion: v1
kind: Pod
metadata:
  name: <Podname>
spec:
  containers:
  - name: <containerName1>
    image: <Image>
    command: <First cmd>
    volumeMounts:                          # Mount definition inside the container
      - name: <volumename
        mountPath: "<Path>"                # Path inside the container to share
  - name: <containerName1>
    image: <Image>
    command: <First cmd>
    volumeMounts:
```

```
    - name: <volumename>
      mountPath: "<Path>”
volumes:                                    # Definition for host
- name: <volumename>
  emptyDir: {}
```