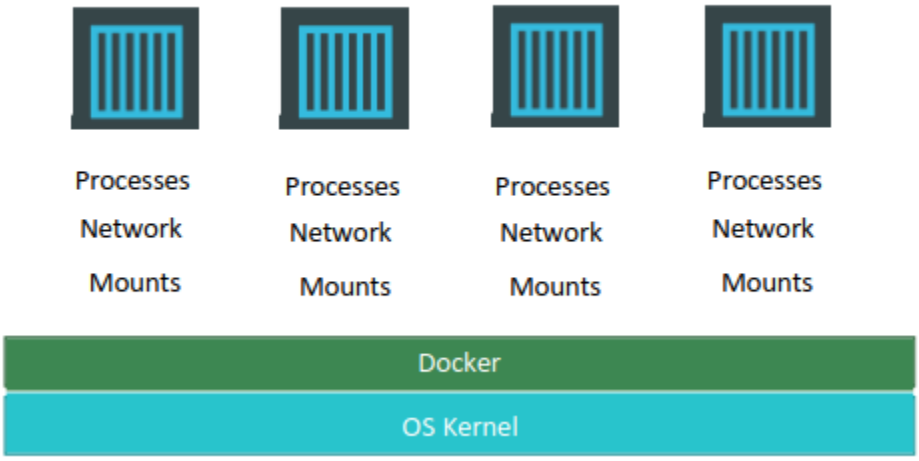
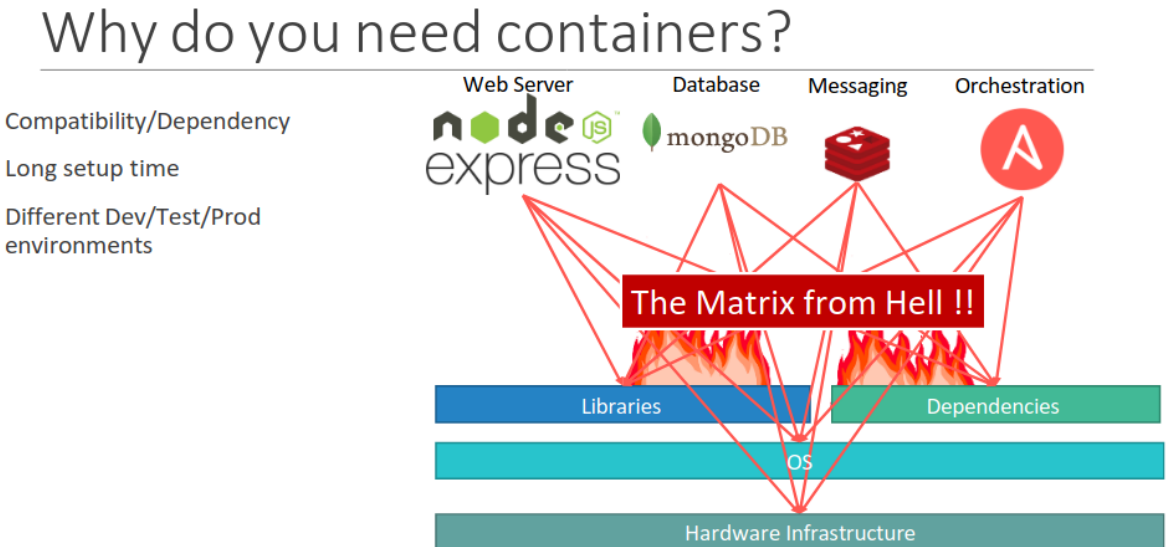
Containers : What and Why?

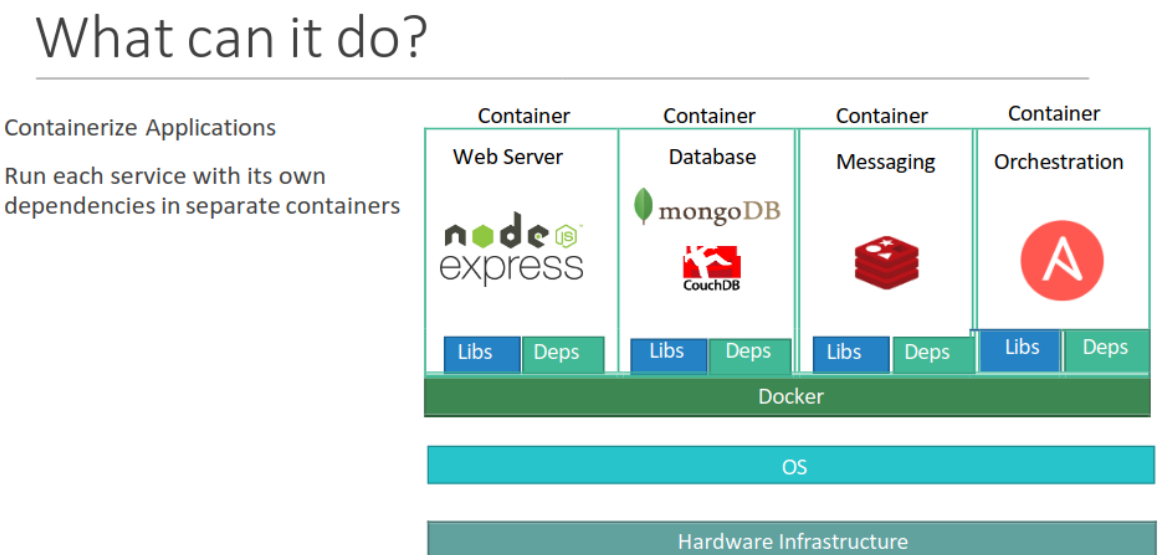


Containers are completely isolated environments, as in they can have their own processes or services, their own network interfaces, their own mounts, just like Virtual machines, except that they all share the same OS kernel. All of the containerized apps share a single, common operating system (either Linux or Windows), but they are compartmentalized from one another and from the system at large. The operating system provides the needed isolation mechanisms to make this compartmentalization happen.



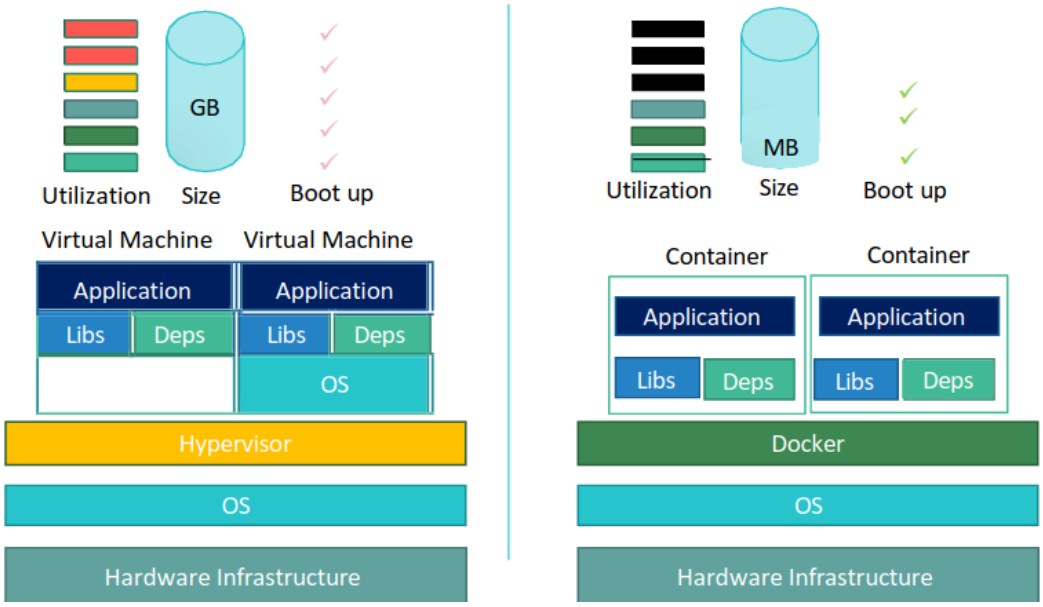
Instances of containerized apps use far less memory than virtual machines, they start up and stop more quickly, and they can be packed far more densely on their host hardware.
Containers make it easy to put new versions of software, with new business features, into production quickly—and to quickly roll back to a previous version if you need to. They also make it easier to implement strategies like blue/green deployments.
containers encapsulate everything an application needs to run (and only those things), they allow applications to be shuttled easily between environments. Any host with the container runtime installed—be it a developer's laptop or a public cloud instance—can run a container.
One of the software patterns containers make easier is microservices
, where applications are constituted from many loosely coupled components. By decomposing traditional, "monolithic" applications into separate services, microservices allow the different parts of a line-of-business app to be scaled, modified, and serviced separately—by separate teams and on separate timelines, if that suits the needs of the business.

Docker



Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows you to run many containers simultaneously on a given host. Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Container Vs Virtual Machines



As you can see on the right, in case of Docker, we have the underlying hardware infrastructure, then the OS, and Docker installed on the OS. Docker then manages the containers that run with libraries and dependencies alone. In case of a Virtual Machine, we have the OS on the underlying hardware, then the Hypervisor like a ESX or virtualization of some kind and then the virtual machines. As you can see each virtual machine has its own OS inside it, then the dependencies and then the application.

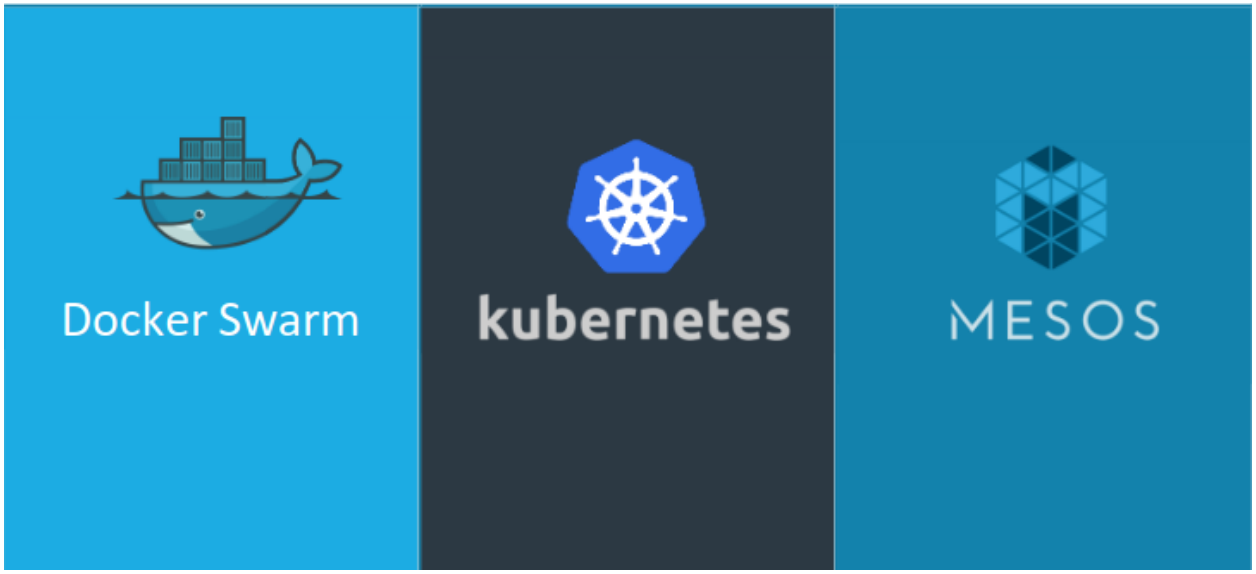
This overhead causes higher utilization of underlying resources as there are multiple virtual operating systems and kernel running. The virtual machines also consume higher disk space as each VM is heavy and is usually in Giga Bytes in size, whereas docker containers are lightweight and are usually in Mega Bytes in size.

Container Orchestration

Process of automatically deploying and managing containers. **Container orchestration** is the automation of much of the operational effort required to run containerized workloads and services. This includes a wide range of things software teams need to manage a container's lifecycle, including provisioning, deployment, scaling (up and down), networking, load balancing and more.

There are various advantages of container orchestration. Your application is now highly available as hardware failures do not bring your application down because you have multiple instances of your application running on different nodes. The user traffic is load balanced across the various containers. When demand increases, deploy more instances of the application seamlessly and within a matter of second and we have the ability to do that at a service level. When we run out of hardware resources, scale the number of nodes up/down without having to take down the application. And do all of these easily with a set of declarative object configuration files.

Orchestration Technologies



There are multiple container orchestration technologies available today – Docker has its own tool called Docker Swarm. Kubernetes from Google and Mesos from Apache. While Docker Swarm is really easy to setup and get started, it lacks some of the advanced autoscaling features required for complex applications. Mesos on the other hand is quite difficult to setup and get started, but supports many advanced features. Kubernetes - arguably the most popular of it all – is a bit difficult to setup and get started but provides a lot of options to customize deployments and supports deployment of complex architectures. Kubernetes is now supported on all public cloud service providers like GCP, Azure and AWS and the kubernetes project is one of the top ranked projects in Github.

Kubernetes

Kubernetes also known as K8s was built by Google based on their experience running containers in production. It is now an open-source project and is the best and most popular container orchestration technologies out there.

Architecture

- Nodes :

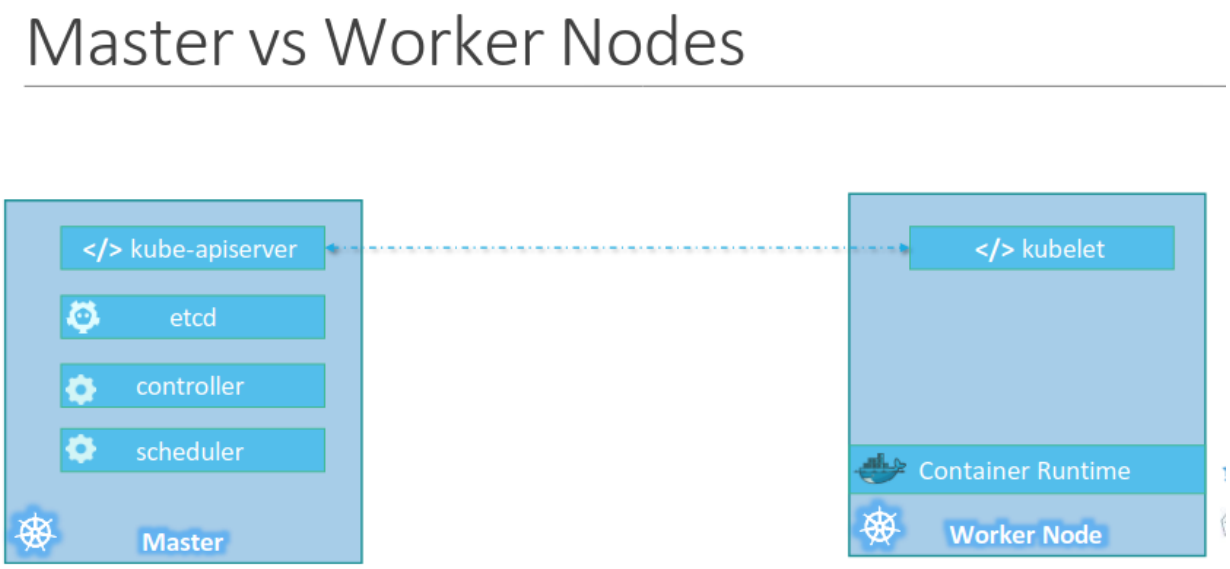
  A node is a machine – physical or virtual – on which kubernetes is installed. A node is a worker machine and this is were containers will be launched by kubernetes.

- Cluster :

  A cluster is a set of nodes grouped together.

- Master :

  The master is another node with Kubernetes installed in it, and is configured as a Master. The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes.



- API Server :

  The API server acts as the front-end for kubernetes. The users, management devices, Command line interfaces all talk to the API server to interact with the kubernetes cluster.

- ETCD :

  ETCD is a distributed reliable key-value store used by kubernetes to store all data used to manage the cluster. Think of it this way, when you have multiple nodes and multiple masters in your cluster, etcd stores all that information on all the nodes in the cluster in a

distributed manner. ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters.

- Scheduler :

  The scheduler is responsible for distributing work or containers across multiple nodes. It looks for newly created containers and assigns them to Nodes.

- Controller :

  The controllers are the brain behind orchestration. They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers makes decisions to bring up new containers in such cases.

- Container Runtime :

  The container runtime is the underlying software that is used to run containers. In our case it happens to be Docker.

- Kubelet :

  kubelet is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected.

Kubectl

The kube control tool is command line tool used to deploy and manage applications on a kubernetes cluster, to get cluster information, get the status of nodes in the cluster and many other things.

Minikube

Different components of Kubernetes that make up a Master and worker nodes are the api server, etcd key value store, controllers and scheduler on the master and kubelets and container runtime on the worker nodes.

Minikube bundles all of the different components into a single image providing us a pre-configured single node kubernetes cluster. The whole bundle is packaged into an ISO image

and is available online for download.

Minikube provides an executable command line utility that will automatically download the ISO and deploy it in a virtualization platform such as Oracle Virtualbox.

Kubernetes using Minikube

Prerequisites :

- Update and upgrade system

  sudo apt update

  sudo apt upgrade

- Install Virtualbox (optional)

  If docker is installed then minikube will work with docker

  Download .deb package - https://www.virtualbox.org/wiki/Linux_Downloads

  sudo dpkg -i <package name>

- Install kubectl

  sudo snap install kubectl --classic

- Install minikube

  curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd64.deb

  sudo dpkg -i minikube_latest_amd64.deb

- Kubernetes extension

  Install Kubernetes extension by microsoft in VSCode.

Start minikube cluster :

minikube start

Start the VM with 4000MB or give custom memory, default is 2GB . -p tag to give a name to the cluster , --nodes tag to give specific number of nodes , --driver tag to specify the driver (can be virtualbox or docker)

minikube start --memory=4000 --nodes=3 --driver=docker -p samplecluster

- See Minikube status

  minikube status

- Check if cluster created and see all objects in the cluster

  kubectl get all

- Minikube dashboard (optional)

  minikube dashboard

  Browser GUI for managing cluster, can be used instead of using kubectl tool

- Stop minikube cluster

  minikube stop

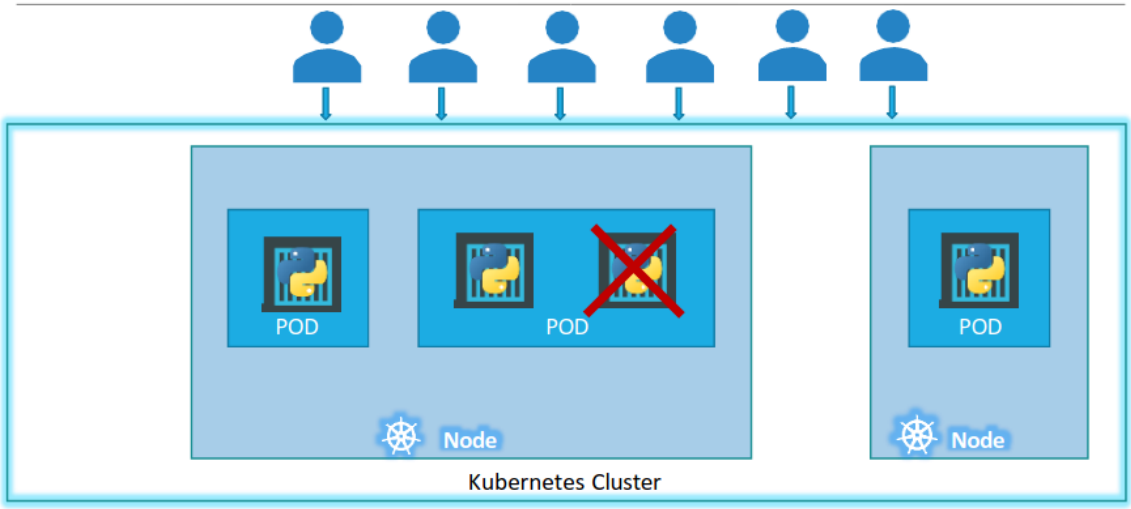
- Destroy cluster and clear all data

  minikube delete

- See the nodes

  kubectl get nodes

POD



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. PODs usually have a one-to-one relationship with containers running your application. 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.
A single POD CAN have multiple containers , except for the fact that they are usually not multiple containers of the same kind. if our intention was to scale our application, then we would need to create additional PODs. But sometimes you might have a scenario were you have a helper container, that might be doing some kind of supporting task for our web application such as processing a user entered data, processing a file uploaded by the user etc. and you want these helper containers to live along side your application container. In that case, you CAN have both of these containers part of the same POD, so that when a new application container is created, the helper is also created and when it dies the helper also dies since they are part of the same POD. The two containers can also communicate with each other directly by referring to each other as 'localhost' since they share the same network namespace. Plus they can easily share the same storage space as well.

POD Demo

Note: in the below commands "po" can be used as shorthand for "pods"

There are 2 ways to create pods.

1. From the command line :

- Create pod

  kubectl run sample --image=nginx

  Here we are using kubectl tool , “sample” is the name given to pod by user and the image we used here is nginx

- See all pods

  kubectl get pods

- Get pod details

  kubectl describe pod sample

- Get more pod details

  kubectl get pods -o wide

- Delete pod

  kubectl delete pod sample

- Delete all pods

  kubectl delete --all pods

2. Using YAML file :

pod-definition.yml

```
apiVersion: v1
kind:
metadata:



spec:
```

Kind	Version
POD	v1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1

Create a file with yaml extension. I naming it pod.yaml

```yaml
apiVersion: v1

kind: Pod

metadata:

 name: mypod

 labels:

   type: frontend

   app: myapp

spec:

 containers:

 - name: myapp

   image: nginx

   resources:

     limits:

       memory: "128Mi"

       cpu: "500m"

   ports:

     - containerPort: 80
```

The POD definition file (pod.yaml) has 4 mandatory fields :

- apiVersion :

This specifies the version of the Kubernetes API being used. As seen in the above file we need to give this according to the object we are creating. As we are creating a pod we use v1

- kind :

  This specifies the kind of Kubernetes resource being defined, which in this case is a Pod.

- metadata :

  This section specifies metadata for the Pod, including its name and labels that can be used to identify or group the Pod with other resources.

  This section has 2 fields :

  - name :

    Specify the name of the pod here. We give the name “mypod”

  - labels :

    Labels are used to organize and select subsets of resources based on their attributes. Labels are used to identify similar or pods under one specific app or performing a specific function. They are like giving tags to group pods in a specific tier. You will understand its need when we go to more complex topics.

    These are user defined and we can provide any key value pair as we like. Here I have given “type: frontend” to just say that this pod is a part of my apps frontend and “app: myapp” this is to specify that this pod is a part of an application called “myapp”.

  - spec:

    Specifies the Pod's specifications.

    - containers :

      Specifies the container(s) to be run in the Pod.

      - name: myapp:

        Specifies the name of the container.

- image: nginx:

  Specifies the Docker image used for the container.

- resources:

  Specifies resource requirements such as CPU and memory limits for the container.

  - limits:

    Defines the maximum amount of resources the container can consume.

    - memory: "128Mi":

      Limits the memory usage to 128 megabytes.

    - cpu: "500m":

      Limits the CPU usage to 500 milliCPU.

- ports:

  Specifies the network ports to be exposed by the container.

  - containerPort: 80:

    Specifies that the container listens on port 80.

This manifest file defines a single container running the nginx image and sets resource limits for the container. It also specifies that the container should listen on port 80. Finally, the Pod is labeled with type: frontend and app: myapp for organizational purposes.

- Create pod using the yaml file

  kubectl create -f pod.yaml

- See all pods

  kubectl get pods

- Group all PODS with same labels

kubectl get pods -l app=myapp

The above command will list all pods with the label “app:myapp” in it.

- Edit pod details

  2 methods:

  1. Edit the .yaml file, save it then

     kubectl apply -f pod.yaml

  2. Edit the pod spec from terminal

     kubectl edit pod mypod

     Opens up vim editor, edit save and exit.

     Note: this doesn’t edit the pod definition (pod.yaml) file.

All commands are the same as earlier mentioned.

Namespace

In Kubernetes, a namespace is a way to create virtual clusters within a physical cluster. It is a logical abstraction layer that allows you to divide your cluster into multiple virtual clusters, each with its own resources, services, and configuration. Namespaces provide a way to organize and isolate resources, as well as to control access to them.

By default, Kubernetes resources are created in the "default" namespace, but you can create additional namespaces to isolate resources and services. For example, you can create a namespace for development, staging, or production environments, or for different teams or applications. This helps in better organization of the resources and services in a large and complex system, and provides better security and isolation.

In addition to organizing resources, namespaces also provide a scope for naming resources. This means that two resources in different namespaces can have the same name without conflicting with each other.

In minikube we will be at default namespace and there will be 3 other namespace created by the tool.

Note: in the below commands “ns” can be used as shorthand for “namespace”

Create a namespace :

- 2 Ways :
    1. Using terminal

       kubectl create namespace sample

    2. Using .yaml file :

```yaml
apiVersion: v1

kind: Namespace

metadata:

 name: sample
```

The fields in the above yaml file are the same as the yaml file for pod creation with change in kind.

- Create namspace using the yaml file

  kubectl create -f namespace.yaml

- See all namespace

  kubectl get ns

- Get namespace (ns) details

  kubectl describe ns sample

- Get more ns details

  kubectl get ns -o wide

- Delete ns

  kubectl delete ns sample

- Create pod in a ns

kubectl create -f pod.yaml --namespace sample

- See pods in a ns

  kubectl get pods --namespace sample

- Delete pod in a ns

  kubectl delete pod mypod --namespace sample

- Delete all pods in a namespace

  kubectl delete --all pods --namespace sample

- See current ns

  kubectl config view --minify | grep namespace:

- Change ns

  kubectl config set-context --current --namespace sample

ReplicaSet

In Kubernetes, a ReplicaSet is a higher-level abstraction that provides a declarative way to ensure a specific number of replicas (pods) for a given Pod template are running at all times.

A ReplicaSet monitors the status of its replicas and ensures that the desired number of replicas is maintained even if a Pod fails or is deleted. It creates new replicas to replace those that have been terminated or removed.

ReplicaSets are typically used to ensure high availability and fault tolerance for stateless applications, such as web servers, where the same workload can be distributed across multiple replicas.

Create a ReplicaSet using .yaml file:

```
apiVersion: apps/v1

kind: ReplicaSet
```

```yaml
metadata:
 name: nginx-replicaset
 labels:
   app: nginx-app
   type: frontend
spec:
 template:
   metadata:
     name: nginx-pod
     labels:
       app: nginx-app
       type: frontend
   spec:
     containers:
       -  name: nginx-container
          image: nginx
 replicas: 5
 selector:
   matchLabels:
     type: frontend
```

This is a YAML file that defines a ReplicaSet in Kubernetes.

- apiVersion: apps/v1:

This indicates the Kubernetes API version and the type of resource that we want to create, which is a ReplicaSet.

- kind: ReplicaSet:

  This defines the type of resource that we are creating, which is a ReplicaSet.

- metadata:

  This section provides metadata about the ReplicaSet such as its name and labels.

  - name: nginx-replicaset:

    This is the name of the ReplicaSet.

  - labels:

    These are the labels attached to the ReplicaSet, which can be used to select and filter this resource.

    - app: nginx-app:

      This is a label with the key "app" and the value "nginx-app".

    - type: frontend:

      This is a label with the key "type" and the value "frontend".

- spec:

  This section defines the desired state of the ReplicaSet.

  - template:

    This section defines the Pod template that will be used to create the replicas.

    - metadata:

      This section provides metadata about the Pod template, such as its name and labels.

      - name: nginx-pod:

        This is the name of the Pod template.

      - labels:

These are the labels attached to the Pod template.

- app: nginx-app:

  This is a label with the key "app" and the value "nginx-app".

- type: frontend:

  This is a label with the key "type" and the value "frontend".

- spec:

  This section defines the specification of the containers that will be run in the Pod.

  - containers:

    This is a list of containers that will be run in the Pod.

    - name: nginx-container:

      This is the name of the container.

    - image: nginx:

      This is the Docker image that will be used for the container.

- replicas: 5:

  This specifies that we want 5 replicas (pods) of the Pod template to be created.

- selector:

  This specifies how the ReplicaSet should select the Pods that it manages.

  - matchLabels:

    This specifies the labels that the Pods must have in order to be selected by the ReplicaSet.

    - type: frontend:

      This is a label with the key "type" and the value "frontend". The ReplicaSet will select Pods that have this label.

Note: in the below commands “rs” can be used as shorthand for “replicaset”

- Create replicaset (rs) using the yaml file

  kubectl create -f replicaset.yaml

- See all rs

  kubectl get rs

- Get rs details

  kubectl describe rs nginx-replicaset

- Get more rs details

  kubectl get rs -o wide

- Delete rs

  kubectl delete rs nginx-replicaset

- Delete all rs

  kubectl delete --all rs

- Scale rs

  2 ways :

  1. Edit .yaml file copy in cluster

     kubectl scale --replicas=6 -f replicaset.yaml

  2. Using command

     kubectl scale rs nginx-replicaset --replicas=8

Deployment

In Kubernetes, a deployment is an abstraction layer on top of the replica set that allows you to declaratively manage a set of replicas of your application. A deployment can be thought of as a

blueprint for creating and managing a set of replicas of your application. It allows you to perform rolling updates, rollbacks, and scaling up or down your application's replicas easily.

With a deployment, you can specify the desired state of your application, including the number of replicas, the container image and its configuration, and the update strategy. Kubernetes will then ensure that the actual state of your application matches the desired state.

Deployments are an important part of Kubernetes because they allow you to manage your application's replicas in a scalable and fault-tolerant way. Instead of manually creating and managing each replica, you can define the desired state of your application in a deployment, and Kubernetes will take care of the rest.

While ReplicaSets are useful for managing the scaling and availability of a set of identical pods, Deployments provide a higher-level abstraction that includes additional functionality for managing updates and rollbacks of application code. Deployments can ensure that a certain number of replicas of a pod are available and updated with the latest version of the application code, and they provide a way to roll back to a previous version in case of issues.

Deployments also manage ReplicaSets underneath them, making it easier to manage and update multiple ReplicaSets at once. Overall, Deployments are a more powerful tool for managing the deployment and update of applications in a Kubernetes cluster, while ReplicaSets are more focused on managing the replication and availability of a specific set of identical pods.

Create a deployment using .yaml file :

```yaml
apiVersion: apps/v1

kind: Deployment

metadata:

 name: nginx-deployment

 labels:

   app: nginx-app

   type: frontend

spec:

 template:
```

```
  metadata:
    name: nginx-pod
    labels:
      app: nginx-app
      type: frontend
  spec:
    containers:
      -  name: nginx-container
         image: nginx
replicas: 6
selector:
  matchLabels:
    type: frontend
```

Everything in the file is same as replicaset except the kind which is deployment here.

Note: in the below commands “deploy” can be used as shorthand for “deployments”

- Create deployment (deploy) using the yaml file

  kubectl create -f deployment.yaml

- See all deployment

  kubectl get deploy

- Get deployment details

  kubectl describe deploy nginx-deployment

- Get more deployment details

  kubectl get deploy -o wide

- Delete deployment

  kubectl delete deploy nginx-deployment

- Delete all deployment

  kubectl delete --all deploy

- Scale deployment

  2 ways :

  3. Edit .yaml file copy in cluster

     kubectl scale --replicas=6 -f deployment.yaml

  4. Using command

     kubectl scale deploy nginx-deployment --replicas=8

Deployment Update Strategies :

1. Recreate

In the context of Kubernetes, recreate is a strategy for performing a deployment update. When a deployment's update strategy is set to Recreate, Kubernetes will stop all of the pods in the deployment, create new pods with the updated configuration, and then delete the old pods.

This approach can result in downtime, as the pods are unavailable during the update process. However, it can also be useful in situations where the update involves major changes, such as changes to the container image or configuration that are incompatible with the existing pods. By creating entirely new pods, Kubernetes can ensure that the update is applied cleanly, without any lingering issues from the old configuration.

2. RollingUpdate

    Rolling update is a strategy used to update applications running on Kubernetes. In this strategy, a new version of the application is gradually rolled out to replace the old version, with a specified number of replicas updated at a time. This process continues until all the replicas are updated to the new version. Rolling updates help to minimize downtime and ensure that the application remains available throughout the update process. If any issues arise during the update, the rollout can be easily rolled back to the previous version.

Rollout

Rollout refers to the process of updating or reverting to a new version of an application in a Kubernetes cluster. It is a crucial part of the deployment process and ensures that the application runs smoothly without any issues.

A rollout involves creating a new version of an application and updating the Kubernetes deployment with the new version. Kubernetes then manages the rollout by creating new replicas of the updated application and gradually scaling down the replicas of the old version until they are replaced completely.

Rollouts can also be used to revert to a previous version of an application if there are issues with the new version. In this case, Kubernetes rolls back to the previous version by scaling up the old replicas and scaling down the new replicas until the old version is fully restored.

Rollback

Rollback is the process of reverting changes made to a system or application to a previous state. In the context of Kubernetes, rollback refers to the process of undoing a deployment to a previous version.

When a deployment is rolled back, Kubernetes replaces the current version of the application with the previous version. This can be useful in scenarios where a new deployment contains a bug or an issue that needs to be resolved. Instead of having to manually undo the changes, Kubernetes provides a simple mechanism to roll back to a previous version of the application.

During a rollback, Kubernetes also updates the deployment's revision number, so that the deployment history is maintained and can be traced.

Check rollout status :

- **kubectl rollout status deploy nginx-deployment**

See rollout history :

- **kubectl rollout history deploy nginx-deployment**

Undo previous rollout ,ie, perform rollback :

- **kubectl rollout undo deploy nginx-deployment**

Edit the deployment and save the edit action (automatic) :

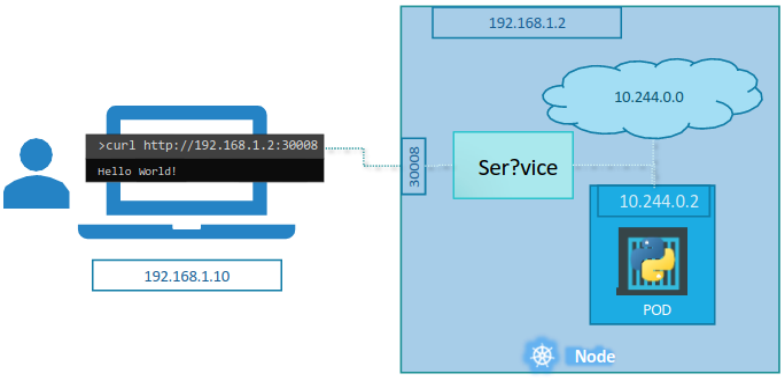
- **kubectl edit deploy nginx-deployment --record**

Edit the deployment and save the edit action (manual) :

- **kubectl edit deploy nginx-deployment**

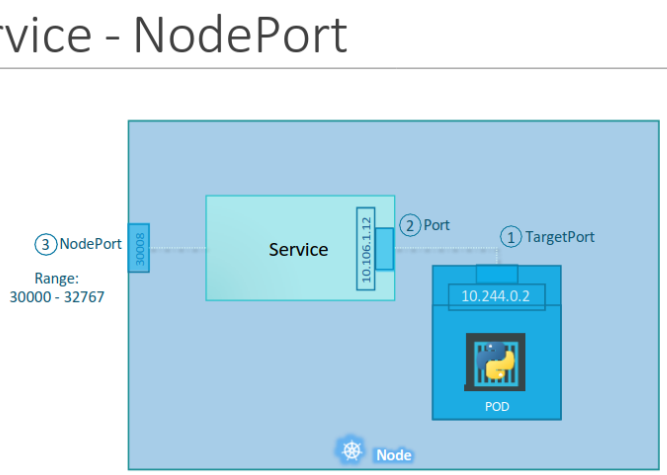
Services

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. For example, our application has groups of PODs running various sections, such as a group for serving front-end load to users, another group running back-end processes, and a third group connecting to an external data source. It is Services that enable connectivity between these groups of PODs. Services enable the front-end application to be made available to users, it helps communication between back-end and front-end PODs, and helps in establishing connectivity to an external data source.



Types :

1. NodePort



A NodePort service in Kubernetes allows access to a specific port on all the worker nodes (a.k.a., the nodes that run the application) in the cluster. The service then forwards the request to the corresponding pod that matches the selector.

When you create a NodePort service, Kubernetes assigns a static port (which is in the range of 30000-32767 by default) to your service, and any incoming traffic that matches the service is forwarded to the backend pods. This port can be used to access the service externally, outside of the cluster, by specifying the worker node's IP address and the assigned static port.

For example, if you create a NodePort service that listens on port 30080, and you have three worker nodes with IP addresses 192.168.1.10, 192.168.1.11, and 192.168.1.12, then the NodePort service will be accessible on all of these nodes on port 30080. To access the service externally, you can use any of the worker node's IP addresses and the assigned static port, like `http://<node-ip>:<nodeport>`.

```yaml
apiVersion: v1
kind: Service
metadata:
 name: my-service-nodeport
spec:
 type: NodePort
 ports:
   - targetPort: 80
     port: 80
```

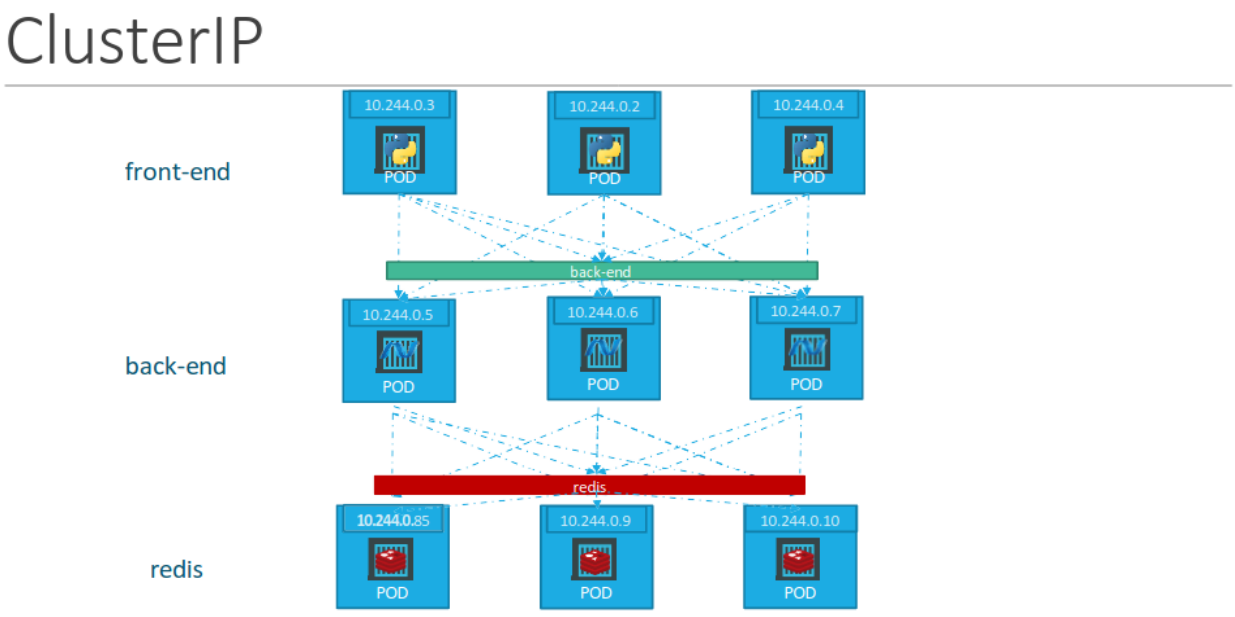
```
    nodePort: 30008
selector:
  app: nginx-app
  type: frontend
```

This code defines a Kubernetes Service named "my-service-nodeport" with a type of "NodePort". The Service is used to expose the pods running the "nginx-app" application as network services within the Kubernetes cluster.

The Service is defined to listen on port 80, and has a nodePort of 30008 which means that the Service will be accessible on each node in the cluster at port 30008. When traffic comes to the node on port 30008, it will be forwarded to the pod on port 80.

The selector field specifies which pods the service should target. In this case, it selects the pods with the label "app=nginx-app" and "type=frontend".

2. ClusterIP



In Kubernetes, a ClusterIP service is a type of service that creates a virtual IP address inside the cluster to allow other objects, such as pods or services, to access it. ClusterIP is the default

service type in Kubernetes, and it is used to provide access to a set of pods from within the cluster.

When you create a ClusterIP service, Kubernetes assigns it a virtual IP address (VIP) that is used by other objects to access the pods associated with the service. The VIP is only accessible from within the cluster, and traffic sent to it is automatically load balanced across the pods.

Overall, ClusterIP services are a useful way to provide access to pods running in a Kubernetes cluster, and they form an important part of the Kubernetes networking stack.

```yaml
apiVersion: v1
kind: Service
metadata:
 name: my-service-clusterip
spec:
 type: ClusterIP
 ports:
   - targetPort: 80
     port: 80
 selector:
   app: nginx-app
   type: frontend
```

This is a Kubernetes YAML manifest for creating a Kubernetes Service resource. The kind specified in the manifest is Service, indicating that this YAML file is for creating a Kubernetes Service resource.

The name of the service resource is specified as my-service-clusterip in the metadata section of the manifest. The spec section specifies the configuration for the service.

This particular service is of type: ClusterIP, which creates a virtual IP address for the service within the Kubernetes cluster.

The ports section specifies the configuration for the ports used by the service. In this case, the port is set to 80, which is the port that the service will listen on, and targetPort is set to 80, which is the port that the backend pods are listening on.

Finally, the selector field specifies that the service will route traffic to pods that have a label with app: nginx-app and type: frontend. This means that only pods with these labels will receive traffic from this service.

Note: in the below commands "svc" can be used as shorthand for "services"

- Create services (svc) using the yaml file

  kubectl create -f services.yaml

- See all services

  kubectl get svc

- Get services details

  kubectl describe svc my-service

- Get more services details

  kubectl get svc -o wide

- Delete service

  kubectl delete svc my-service

- Delete all services

  kubectl delete --all svc

- If nodePort service is up then we can access the pods from external browser or terminal

  minikube service my-service-nodeport --url

  OR

  http://<ip address of node>:<nodeport>

Persistent Volume & Persistent Volume Claim

In Kubernetes, a Persistent Volume (PV) is a piece of storage that has been provisioned by an administrator. It is a way of decoupling storage configuration from the container and Pod definition. It provides a way for data to persist across the lifecycle of a Pod.

A Persistent Volume is an object in the Kubernetes API that represents a piece of networked storage. The storage can be provided by many different storage systems like local disks, network-attached storage (NAS), cloud storage, or a host directory.

Once a PV is created, it can be used by a Pod by defining a Persistent Volume Claim (PVC). A PVC is a request for storage that matches specific requirements such as size, access mode, and storage class. When a PVC is created, Kubernetes will find a PV that matches the requirements and bind it to the PVC. The Pod can then use the PVC as a volume to store data.

In summary, a Persistent Volume is a way to provision and manage storage in Kubernetes. It provides a way for data to persist across the lifecycle of a Pod.

Create a pv using .yaml file

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
 name: mypv
spec:
 capacity:
   storage: 500Mi
 volumeMode: Filesystem
 accessModes:
   - ReadWriteOnce
 persistentVolumeReclaimPolicy: Recycle
 storageClassName: slow
 hostPath:
   path: "/myStorage"
```

- apiVersion: Indicates the Kubernetes API version to use for this object.

- kind: Indicates the type of object, in this case a Persistent Volume (PV).

- metadata: Specifies the name of the PV and any additional metadata such as labels.

- spec: Specifies the details of the PV such as capacity, access modes, and storage class.

Here is a breakdown of the spec section:

- capacity: Specifies the amount of storage the PV should provide.

- volumeMode: Specifies the type of volume to create. In this case, it is a filesystem.

  When volumeMode is set to Filesystem, it means that the volume is formatted with a file system, like ext4 or NTFS, and can be mounted as a regular file system within the container. This is the default mode for volumes and is suitable for most applications that need to store and read data in a traditional file system.

  Other supported values for volumeMode include Block and Raw, which are used when a volume is used as raw block storage rather than a file system.

- accessModes: Specifies the access modes for the PV. In this case, it can only be mounted by one node in read-write mode.

  Other possible values for accessModes are:

  - ReadOnlyMany: the volume can be mounted as read-only by multiple nodes.
  - ReadWriteMany: the volume can be mounted as read-write by multiple nodes.

- persistentVolumeReclaimPolicy:

  Specifies the policy for what to do with the volume after it is released. In this case, the policy is set to Recycle which means that Kubernetes will delete the contents of the volume when it is released.

persistentVolumeReclaimPolicy is a setting in Kubernetes that defines what should happen to the persistent volume (PV) when the claim to which it is bound is deleted.

Recycle is one of the possible values for this setting, indicating that the PV should be deleted and its contents recycled. This means that when a user deletes the claim that was bound to the volume, Kubernetes will automatically delete the volume and, if there is any data on it, overwrite that data with zeroes. This makes the volume available for reuse by new claims.

It's worth noting that Recycle is deprecated as of Kubernetes v1.21 and will be removed in a future release. Users are encouraged to use other reclaim policies like Retain or Delete.

- storageClassName: Specifies the storage class to use for the PV.

  In Kubernetes, a StorageClass is used to define the different types of storage that can be used in a cluster. It allows you to set the type of storage, the access modes, and the reclaim policy for the storage. In the given code storageClassName: slow is set, which indicates that the StorageClass named slow is used for the PersistentVolume. This could mean that the StorageClass provides storage with lower performance characteristics, such as lower IOPS or throughput, or maybe it is simply a name chosen to indicate that this storage should be used for applications that do not require high performance. The exact properties of the storage provided by the slow StorageClass would be defined in the corresponding StorageClass object.

  There are several pre-defined storage class names in Kubernetes that serve different purposes. Here are some common storage class names:

  - `standard`: This is the default storage class in Kubernetes, and is suitable for general-purpose use cases.
  - `slow`: This storage class is typically used for non-critical data storage, such as backups or logs.
  - `fast`: This storage class is optimized for high-performance workloads, such as databases or analytics.
  - `local-storage`: This storage class is used for attaching local storage devices to a node, rather than using a networked storage system.
  - `aws-ebs`: This storage class is used for provisioning Amazon Elastic Block Store (EBS) volumes in AWS environments.
  - `azure-disk`: This storage class is used for provisioning Azure Disk volumes in Microsoft Azure environments.

- `gce-pd`: This storage class is used for provisioning Google Compute Engine (GCE) persistent disks in GCP environments.

- hostPath: Specifies that this PV uses a host path for storage. The path field specifies the directory on the host where the storage is located.

  In Kubernetes, hostPath is used to access a directory or file on the host node's file system from a pod. The hostPath volume mounts a file or directory from the host node's filesystem into a pod.

  In the code snippet hostPath is used to define the location of the directory that will be mounted as a volume in a Pod. Specifically, it specifies the path on the host's file system where the volume data is stored.

  In this example, the directory /myStorage on the host node's file system will be mounted as a volume in a pod that uses this PersistentVolume. Any data stored in this directory will be accessible to the pod.

Create a pvc using .yaml file

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: mypvc
spec:
 volumeName: mypv
 storageClassName: slow
 resources:
   requests:
     storage: 300Mi
 volumeMode: Filesystem
 accessModes:
   - ReadWriteOnce
```

This YAML code is defining a PersistentVolumeClaim (PVC) named "mypvc" with the storage class "slow". It requests a storage of 300Mi from the PersistentVolume (PV) named "mypv" with the same storage class.

Here is a breakdown of the different parts of this code:

- apiVersion: v1: specifies the API version being used
- kind: PersistentVolumeClaim: specifies that this YAML code defines a PersistentVolumeClaim object
- metadata: contains metadata of the PVC object, such as its name
- spec: contains the specifications for the PVC object, including its volume name, storage class, resource requests, access modes, and volume mode
    - volumeName: specifies the name of the PersistentVolume that this PVC is bound to, in this case, "mypv"
    - storageClassName: specifies the storage class for the PVC, which should match the storage class of the PV it is bound to
    - resources: specifies the resources requested by the PVC, in this case, 300Mi of storage
    - accessModes: specifies the access modes requested by the PVC, in this case, ReadWriteOnce, which means the volume can be mounted as read-write by a single node
    - volumeMode: specifies the volume mode of the PVC, which is Filesystem by default.

Overall, this YAML code creates a PVC that requests 300Mi of storage from a specific PV named "mypv", with the "slow" storage class, and the ability to be mounted as read-write by a single node.

Add the pvc to deployment definition file

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: my-deployment
 labels:
   app: nginx-app
   type: frontend
spec:
 template:
   metadata:
     name: nginx-pod
     labels:
       app: nginx-app
       type: frontend
   spec:
```

```
    volumes:
      - persistentVolumeClaim:
          claimName: mypvc
        name: myvolume
    containers:
      -  name: nginx-container
         image: nginx
         volumeMounts:
           - name: myvolume
             mountPath: "/kba"
replicas: 6
selector:
  matchLabels:
    type: frontend
```

This YAML code creates a Deployment object named "my-deployment" that manages a set of replicas of a Pod. The Pod is created based on the template specified in the Deployment's specification. The template defines a Pod with the name "nginx-pod" and labels "app: nginx-app" and "type: frontend".

The Pod specification includes a volume named "myvolume" that is backed by a PersistentVolumeClaim named "mypvc". This means that the Pod can use the storage defined in the PersistentVolumeClaim. The Pod also includes a container named "nginx-container" that uses the Nginx image and mounts the "myvolume" volume at the "/kba" directory inside the container.

The Deployment specifies that 6 replicas of the Pod should be created, and the replicas are selected based on the "type: frontend" label. This means that any Pod with this label will be managed by this Deployment. The Deployment also includes labels "app: nginx-app" and "type: frontend" for easy identification and organization.

Note: in the below commands are same for pv and pvc just change the keyword accordingly

- Create pv using the yaml file

  kubectl create -f pv.yaml

- See all pv

  kubectl get pv

- Get pv details

  kubectl describe pv mypv

- Get more pv details

  kubectl get pv -o wide

- Delete service

  kubectl delete pv mypv

- Delete all pv

  kubectl delete --all pv

- Enter node shell

  minikube ssh

- Enter pod shell

  kubectl exec -it mypod -- bash

Jobs

In Kubernetes, Jobs are used to run one or more Pods to completion. A Job creates a Pod which runs a specified container and terminates successfully or fails, and then it is restarted until a specified number of successful completions are reached.
Jobs are commonly used for running batch processes, running ETL tasks, running backup jobs, etc. They provide the ability to perform a one-off task in the cluster and guarantee that the task completes successfully before terminating.

Here are some important features of Kubernetes Jobs:

- It guarantees the successful completion of a task before the Pod is terminated
- It can run multiple Pods concurrently
- It can be configured to restart a failed Pod until a specified number of successful completions are reached
- It can be scheduled to run at a specific time or on a specific schedule
- It can be configured to create a CronJob that runs on a regular schedule.

When a Job completes, it leaves behind a Pod, which can be viewed or examined for debugging purposes.

Lens IDE

Lens is an IDE for visualizing kubernetes clusters and objects.

Lens IDE installation :

sudo apt update

sudo apt upgrade

sudo snap install kontena-lens --classic

Open lens ide -> cluster -> + -> add config file from ./kube folder… mostly the file will be shown in the ide just select it. See the nodes, pv, pvc, pods etc in IDE