Minikube is a tool that enables you to run a single-node Kubernetes cluster on your local machine. It provides a lightweight and easy-to-use way to test and develop applications that are designed to run on a Kubernetes cluster. With Minikube, you can quickly create and manage a local Kubernetes cluster, test your applications in a development environment, and simulate production scenarios on your own machine.

Minikube runs a virtual machine on your local machine using a hypervisor such as VirtualBox or Hyper-V, and deploys a single-node Kubernetes cluster inside that virtual machine. This allows you to run a complete Kubernetes environment on your local machine, without the need for a full-scale production cluster.

**SETUP WITH MINIKUBE:**

1. Open Powershell as Admin
2. Setup Chocolaty
3. Install Minikube with Chocolaty

choco install minikube kubernetes-cli

1. Open Powershell and run

minikube start

USING KOPS TO DEPLOY MINIKUBE ON AWS: (Prerequisites)

* Domain for kubernetes DNS records

e.g. groophy.in from GoDaddy

* Create a linux VM and setup (I will use EC2 but Vagrant can also be used)

kops, kubectl, ssh keys, awscli

* Login to AWS account and setup

s3 bucket, IAM User for AWSCLI, Route53 Hosted Zone.

MINIKUBE CONFIGURATION STEPS:

Use the Documentation: kubernetes.io/docs/setup/learning-environment/minikube/

1. Open GitBash
2. git clone https://github.com/devopshydclub/vprofile-project.git
3. cd vprofile-project/
4. git pull (To get the latest changes)
5. git checkout kubernetes-setup
6. ls
7. cd minikube/
8. ls
9. cat Minikube-commands.txt
10. Open Powershell as Administrator and follow the steps in the txt file
11. Run the first command:

Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))

1. If you get a not allowed error, turn off your Antivirus from settings and repeat the steps
2. OR: choco upgrade chocolatey (To upgrade if this is the message)
3. When completed close powershell and open again with Admin privilege
4. Run the second command:

choco install minikube kubernetes-cli -y

1. When completed you are free to close the powershell
2. return to gitbash
3. cd
4. minikube.exe - -help
5. Use the Documentation: kubernetes.io/docs/setup/learning-environment/minikube/
6. minikube start
7. it will take some time, becos it will download an image and bring up a VM in virtualbox
8. when you see your VM, check GitBash for completion status
9. kubectl get nodes (kubectl is using a file which is stored in the Home directory)
10. cat **.**kube/config (kubectl get nodes uses these config file to access the cluster)
11. kubectl create deployment hello-minikube - -image=k8s.gcr.io/echoserver:1.10 deployment.apps/hello-minikube created
12. kubectl.exe get pod
13. kubectl.exe get deploy
14. kubectl expose deployment hello-minikube - -type=NodePort - -port=8080 service/hello-minikube exposed
15. minikube service hello-minikube - -url (To get the URL to access the node)
16. copy the url
17. Go to your browser and paste to access
18. kubectl.exe get svc
19. To delete and Clean up:
20. kubectl.exe delete svc hello-minikube
21. kubectl.exe get deploy
22. kubectl.exe delete deploy hello-minikube
23. minikube.exe stop
24. minikube.exe delete

**STEPS: SETUP WITH KOPS (PREREQUISITES): USING EC2**

1. Login to your AWS account and go to EC2 instances
2. Click on Launch an Instance

* AMI = Ubuntu server 20.04
* instance type = t2.micro
* Number of instances: 1
* Security group = create a security group, name = kops-sg, SSH / TCP / 22 / MyIP
* Keypair = kops-key, download the .csv
* 8GB SSD
* key = name & value = kops
* click on launch

1. Create an S3 Bucket:

* search for S3 Buckets under Services
* click on create a bucket:
* Bucket Name = vprofile-kops-state
* select your EC2 region
* click on create

1. Create an IAM User for AWSCLI & other services (S3):
   * search for IAM under services
   * click on Add User

* user name = kopsadmin
* check programmatic access
* click on Attach existing policies directly
* check on AdministratorAccess
* click on next and click on create
* download the .csv and be very careful with it

1. Create Route53:
   * search for Route 53 under services
   * click on create hosted zone
2. Domain name = kubevpro.groophy.in
3. check Public hosted zone
4. click on Create hosted Zone
5. Scroll down and take note of the Value/Route traffic to info
6. login to your Domain name host provider (truehost)
7. Go to DNS settings / NS Records
8. Type = Nameserver, Host = kubevpro (Domain name provided above), Points to = take the first value in step d above)
9. click on save

* Type = Nameserver, Host = kubevpro (Domain name provided above), Points to = take the second value in step d above)
* click on save
* Repeat step and Do for the third and forth values

1. SSH to your EC2 instance (KOPS) to setup everything:

* Configure the AWSCLI:

1. Go to GitBash
2. cd
3. ssh –i Downloads/kops-key.pem ubuntu@public IP
4. Generate the SSH key to be used by AWSCLI & KOPs
5. ssh-keygen
6. hit enter to accept defaults in the 3 question areas
7. save the Access and Secret keys in word file
8. Install AWSCLI
9. sudo apt update && sudo apt install awscli –y
10. configure the AWSCLI
11. aws configure
12. provide the access and secret keys
13. Region of your EC2
14. Default output = json

* Configure the KOPS: (Use the documentation as a guide: how to install Kubernetes with KOPS)

1. curl –LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
2. ls (To check the Kubectl binary)
3. chmod +x ./kubectl
4. sudo mv kubectl /usr/local/bin
5. curl –LO https://github.com/kubernetes/kops/releases/download/$(curl –s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag\_name | cut –d ‘”’ –f 4)/kops-linux-amd64
6. ls
7. sudo chmod +x kops-linux-amd64
8. sudo mv kops-linux-amd64 /usr/local/bin/kops
9. kops - -help
10. nslookup –type=ns kubevpro.groophy.in
11. kops command to create the kubernetes cluster:
12. kops create cluster - -name=kubevpro.groophy.in \ (press Enter)

> - -state=s3://vprofile-kop-states - -zones= us-east-2a, us-east-2b \ (press Enter)

> - -node-count=2 - -node-size=t3.small - -master-size=t3.medium - -dns-zone=kubevpro.groophy.in \ (press Enter)

> - -node-volume-size=8 - -master-volume-size=8 (press Enter to start creating)

1. configure the cluster: The command will show immediately after last command
2. kops update cluster - -name kubevpro.groophy.in - -state=s3://vprofile-kop-states - -yes - -admin (remember to include your s3 bucket)
3. kops validate cluster - -state=s3://vprofile-kop-states
4. cat ~/.kube/config (These file is used by kubectl to connect to the clusters)
5. kubectl get nodes (You can confirm by checking the EC2 (3ec2), Autoscaling groups (3groups), VPC and Route 53)
6. If you are not using the Cluster, Please stop to avoid Billing:

kops delete cluster - -name=kubevpro.groophy.in - -state=s3://vprofile-kop-states - -yes

1. sudo poweroff

**KUBERNETES OBJECTS AND DOCUMENTATION :**

K8s Objects are:

Here are some of the key Kubernetes objects:

1. Pod: The smallest deployable unit in Kubernetes, representing one or more containers that run together on a single node.
2. Service: An abstraction that defines a logical set of Pods and a policy for accessing them.
3. Deployment: A declarative way to manage Pods and their replica sets, ensuring that a specified number of replicas are running at all times.
4. StatefulSet: Similar to a Deployment, but designed for stateful applications that require stable network identities and stable storage.
5. ConfigMap: A way to manage configuration data as key-value pairs or files.
6. Secret: A way to manage sensitive data, such as passwords or API keys.
7. Volume: A way to provide persistent storage for containers.
8. Namespace: A way to logically partition a cluster into multiple virtual clusters, each with its own set of resources and policies.
9. ServiceAccount: An identity that a Pod can use to authenticate with other parts of the cluster, such as the Kubernetes API server.

These are just a few examples of the many Kubernetes objects that are available. Each object has its own set of configuration options, which allow you to define the desired state of your cluster and manage it in a declarative way.

**The Course looked at: Namespace, Pod, Service, Replica Set, Deployment, Config Map, Secret, Volumes**

**NAMESPACE:** (Objects in K8s) Use the Documentation: kubernetes.io/docs as a guide

In Kubernetes, a Namespace is a way to logically partition a cluster into multiple virtual clusters, each with its own set of resources and policies. Namespaces provide a way to organize and isolate resources, and they allow multiple teams or applications to share a cluster without interfering with each other.

You have to specify your namespace in your kubeconfig file

1. kubectl get ns
2. kubectl get all
3. kubectl get all - -all-namespaces (Shows all the resources in all the namespaces)
4. kubectl get svc –n kube-system (To get the resource of a particular service)
5. kubectl create ns kubekart (To create/specify your own namespace)
6. kubectl run nginx1 - -image=nginx –n kubekart (Pod in the same name but different namespace)

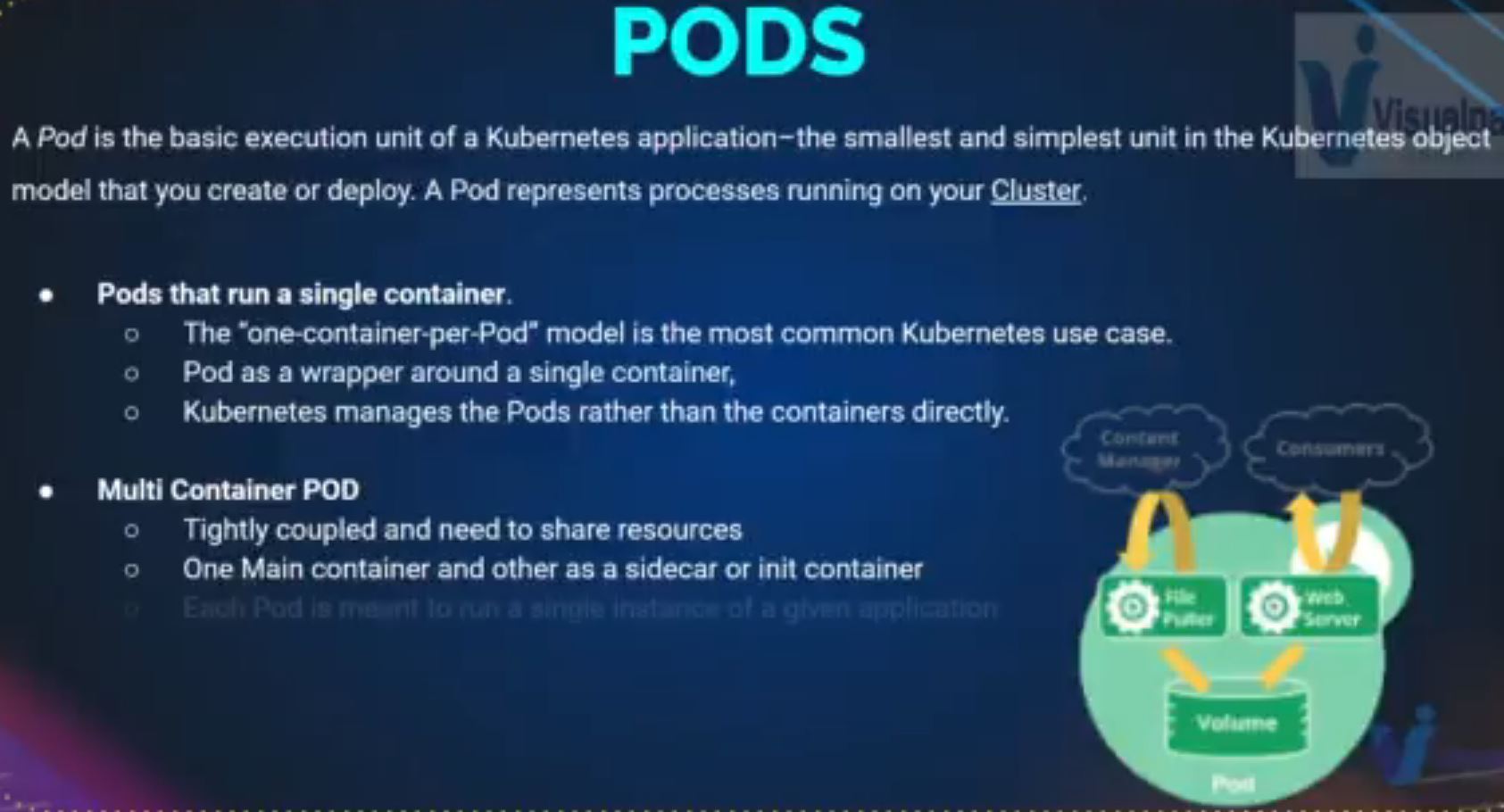
**PODS:** (Objects in K8s) Use the Documentation: kubernetes.io/docs as a guide

In Kubernetes, a Pod is the smallest deployable unit that represents a single instance of a running process in a cluster. A Pod can contain one or more tightly-coupled containers that share the same network namespace, storage volumes, and specifications for how to run the containers.

Pods are the basic building blocks of Kubernetes applications, and they provide a way to encapsulate and manage one or more related containers as a single unit. For example, a Pod might contain a container that runs a web server, along with a sidecar container that handles logging or monitoring.

Here are some key features of Pods:

1. Single-instance unit: A Pod represents a single instance of a running process in a cluster.
2. Shared context: All containers within a Pod share the same network namespace and can communicate with each other using localhost.
3. Shared storage: Containers within a Pod can share the same storage volumes, allowing them to share data.
4. Atomic scheduling: Kubernetes schedules and manages Pods as a single unit, ensuring that all containers within a Pod are scheduled and deployed together.
5. Lifecycles: Pods have their own lifecycle, including creation, deletion, and scaling.



Definitions file in YAML (pod-setup.yml)

Open your code editor .e.g. Vs Code and save the file as: pod-setup.yml

apiVersion: v1

kind: Pod

metadata:

name: webapp-pod

labels:

app: frontend

project: infinity

spec:

containers:

- name: httpd-container

image: httpd

ports:

- name: http-port

containerPort: 80

To Create and get POD Info:

* Go to GitBash
* kubectl create –f pod-setup.yml pod/webapp-pod created
* kubectl get pod (To get information about the POD)
* kubectl describe pod webapp-pod (To get more information about the POD) OR: kubectl describe pod webapp-pod -o yaml (It is more readable)

To Get and Edit a POD

* kubectl get pod webapp-pod –o yaml
* kubectl get pod webapp-pod –o yaml > webpod-definition.yml
* kubectl edit pod webapp-pod

**PROJECT: DEPLOYMENT WITH TOMCAT CONFIG**

1. Create & Connect to the Kops EC2 instance
2. SSH to the instance and sudo -i
3. kops create cluster - -name=kopper.groophy.in - -state=s3://kopper-kops - -zones=us-east-2a, us-east-2b - -node-count=2 - -node-size=t2.micro - -master-size=t2.micro - -dns-zone=kopper.groophy.in
4. kops update cluster - -name kopper.groophy.in - -yes - -state=s3://kopper-kops
5. kops validate cluster - -name kopper.groophy.in - -state=s3://kopper-kops
6. kubectl get nodes
7. ls –a
8. cat .kube/config
9. mkdir definitions
10. cd definitions/
11. mkdir pod
12. cd pod/
13. I would run Tomcat POD first:
14. vim vproapppod.yaml

apiVersion: v1

kind: Pod

metadata:

name: vproapp

labels:

app: vproapp

spec:

containers:

- name: appcontainer

image: imranvisualpath/freshtomapp:V7

ports:

- name: vproapp-port

containerPort: 8080

:wq

1. kubectl create –f vproapppod.yaml
2. kubectl get pod
3. kubectl describe pod vproapp

**DIFFERENT LEVELS OF LOGGING: (Solving & Resolving Errors)**

Ideally when working on a project, it is best you follow these road map:

Local Environment setup (personal Vm or EC2) 🡺 Test in the Local Environment 🡺 Then in the test or Dev Environment 🡺 Then finally in the Production Environment

* kubectl get pod OR:
* kubectl get pod -o wide OR: you specify the image name
* kubectl get pod nginx12 –o yaml 🡺 Under Status if you see: ImagePullBackOff as an error it means you have given a wrong Image name
* So delete the pod
* kubectl delete pod nginx12
* Fix the mistake
* vim pod2.yaml (Give the right Image name and save-quit)
* kubectl apply –f pod2.yaml
* kubectl get pod 🡺 Under Status if you see: CrashLoopBackOff as an error it means restart failed
* kubectl get pod web2 –o wide
* kubectl get pod web2 –o yaml (Check under waiting/message, it would explain the error)
* kubectl get pod (Under Restart you will see the number of restart attempts)
* kubectl describe pod web2 (More explanation for the failure)
* kubectl logs web2
* Fix the mistake
* kubectl delete pod web2
* kubectl run web2 - -image=nginx
* kubectl get pod

**SERVICE:** (Objects in K8s)

In Kubernetes, a Service is an abstraction that defines a logical set of Pods and a policy for accessing them. Services provide a stable IP address and DNS name that can be used to access the Pods, even as the set of Pods changes over time.

Services enable communication between different parts of an application, both within the same cluster and across different clusters. For example, a Service might represent a group of database Pods, and provide a stable IP address and DNS name that can be used by other parts of the application to access the database.

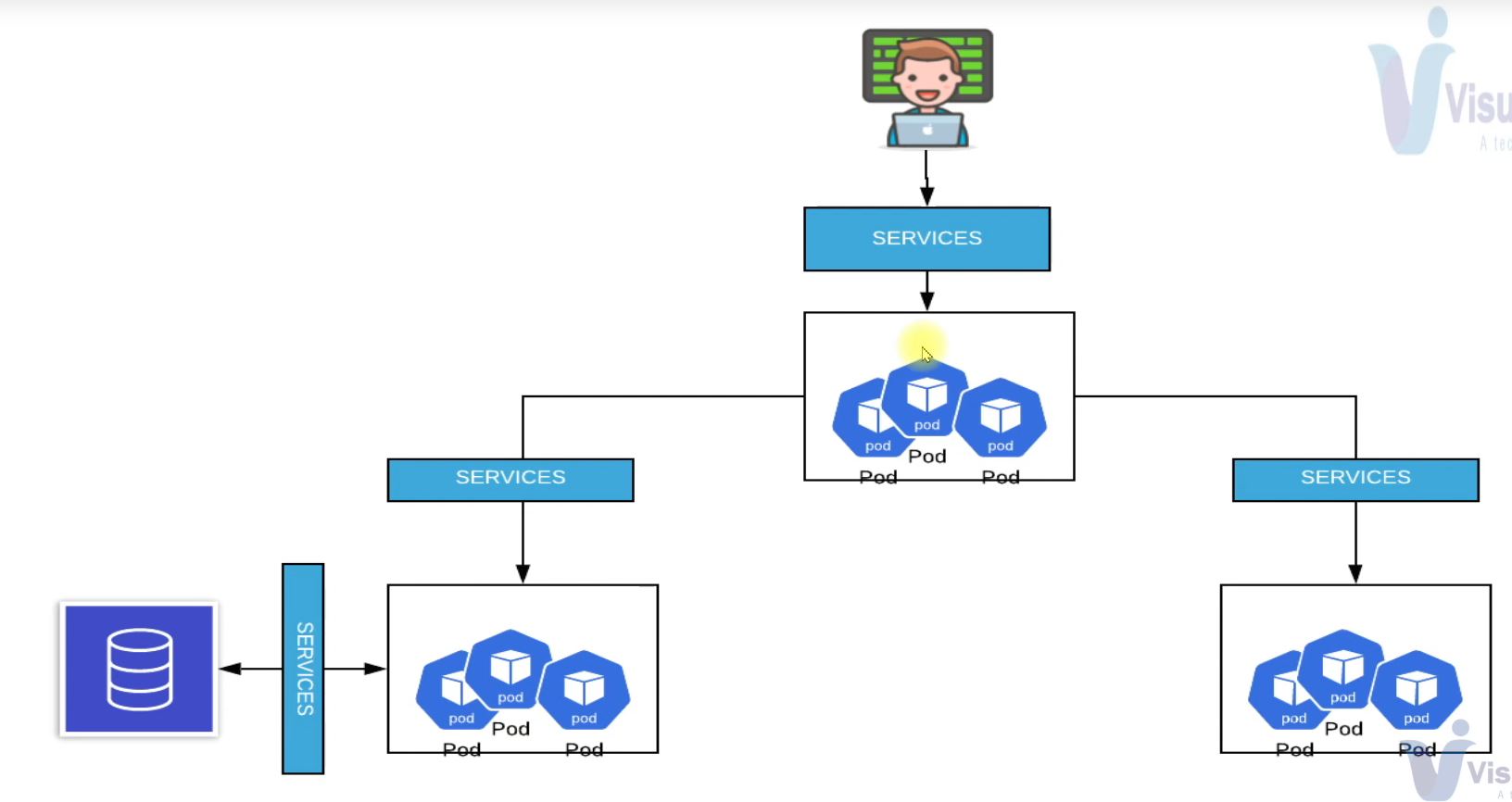
Here are some key features of Services:

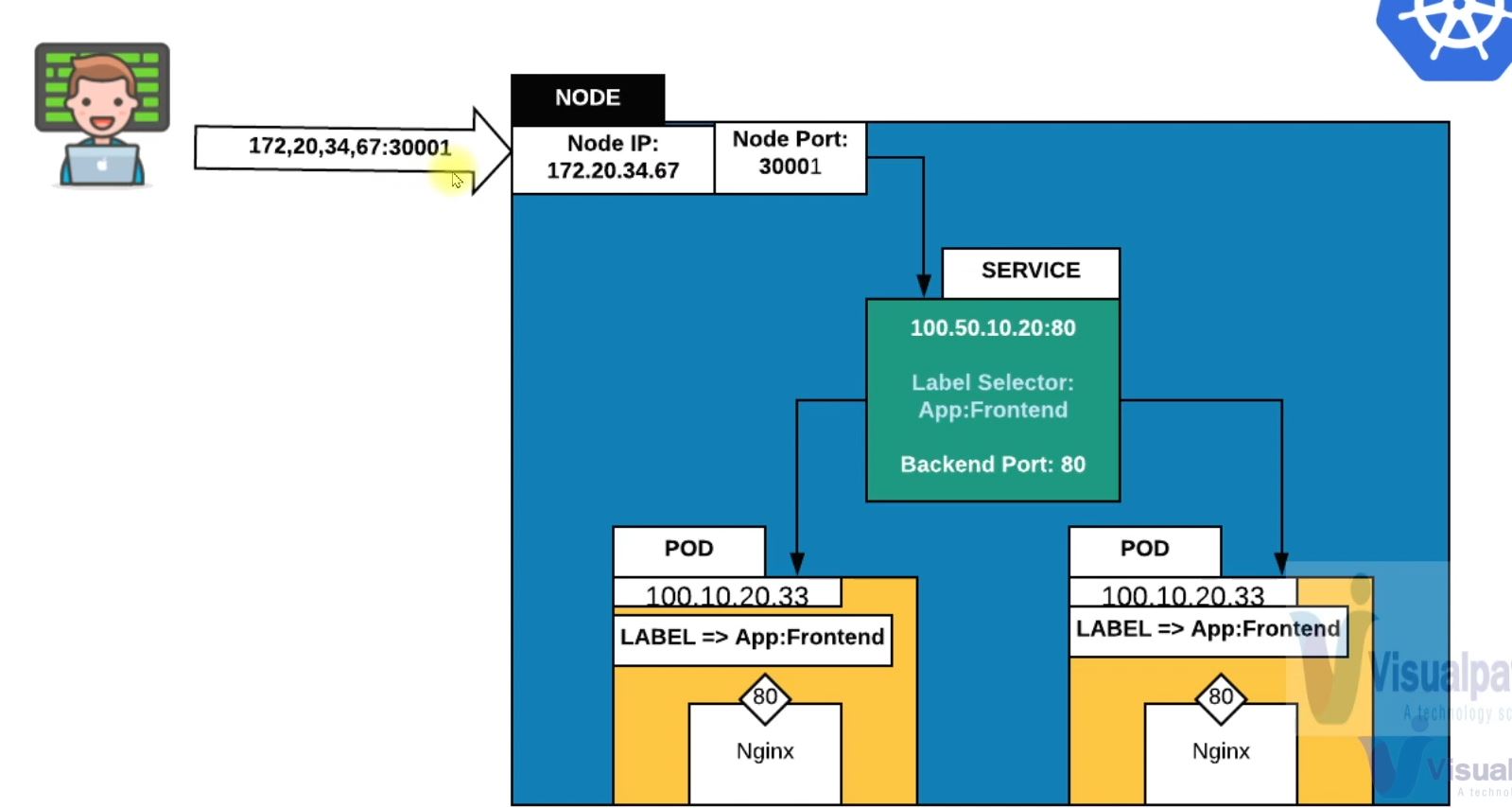
1. Load balancing: Services distribute traffic across a set of Pods, ensuring that no single Pod is overwhelmed with traffic.
2. Service discovery: Services provide a stable IP address and DNS name that can be used to access the Pods, even as the set of Pods changes over time.
3. Cluster-wide access: Services can be accessed from anywhere within the cluster, as long as the appropriate permissions are in place.
4. External access: Services can be exposed to the outside world using Kubernetes Ingress, NodePort, or LoadBalancer resources.

Overall, Services provide a powerful abstraction for managing network traffic within a Kubernetes cluster, enabling communication between different parts of an application and allowing you to expose your services to the outside world.

**TYPE OF SERVICE:**

* NodePort: Very similar to Port Mapping in Docker .i.e. Host port mapped with container port to expose to the outside network
* Cluster IP: If you don’t want to expose it to outside network but want an Internal communication only .e.g. Tomcat - MySQl connection
* LoadBalancer: Used to expose a port to the outside network for Production use case. .e.g. You want User to access the Tomcat service from outside, we need to create a LoadBalancer, on AWS it would create an Elastic LoadBalancer and Map the POD to it





Format for Creating a Service:

apiVersion: v1

kind: Service

metadata:

name: webapp-service

spec:

type: NodePort

ports:

- targetPort: 80

port: 80

nodePort: 30005

protocol: TCP

selector:

app: frontend

**HOW TO CREATE A SERVICE IN FRONT OF TOMCAT: (NodePort / LoadBalancer)**

**NodePort (Type of Service)**

1. sudo –i
2. you should have the definition file
3. cd definitions/pod/ (The folder was created above)
4. cat vproapppod.yaml
5. cd .. (To move one level up into definitions folder)
6. mv pod app
7. cd app/
8. Inside the App directory we should have the service definition
9. cat vproapppod.yaml (To check the config – for the Label & Port no)
10. Let’s write the definitions file
11. vim vproapp-nodeport.yaml

apiVersion: v1

kind: Service

metadata:

name: helloworld-service

spec:

ports:

- targetPort: vproapp-port

port: 8090 (Internal frontend port)

nodePort: 30001 (External frontend port)

protocol: TCP

selector:

app: vproapp

type: NodePort

:wq

1. kubectl create –f vproapp-nodeport.yaml
2. check for detailed info
3. kubectl describe svc helloworld-service
4. kubectl describe pod | grep IP
5. change the security group of the node
6. All traffic / My IP
7. It doesn’t matter which Node the PODs are running on, we would access through the service becos it runs on all Nodes and Master Node also….. refer to the blue diagram above
8. Go to the browser
9. EC2 public IP : 30001
10. For clean up
11. kubectl delete svc helloworld-service

**LoadBalancer: (Type of Service)**

1. Now i would create a service of type LoadBalancer
2. cd definitions/app/
3. ls
4. cp vproapp-nodeport.yaml vproapp-loadbalancer.yml
5. vim vproapp-loadbalancer.yml

apiVersion: v1

kind: Service

metadata:

name: helloworld-service

spec:

ports:

- targetPort: vproapp-port

port: 80 (frontend port) NodePort will be created automatically by default randomly form its range

protocol: TCP

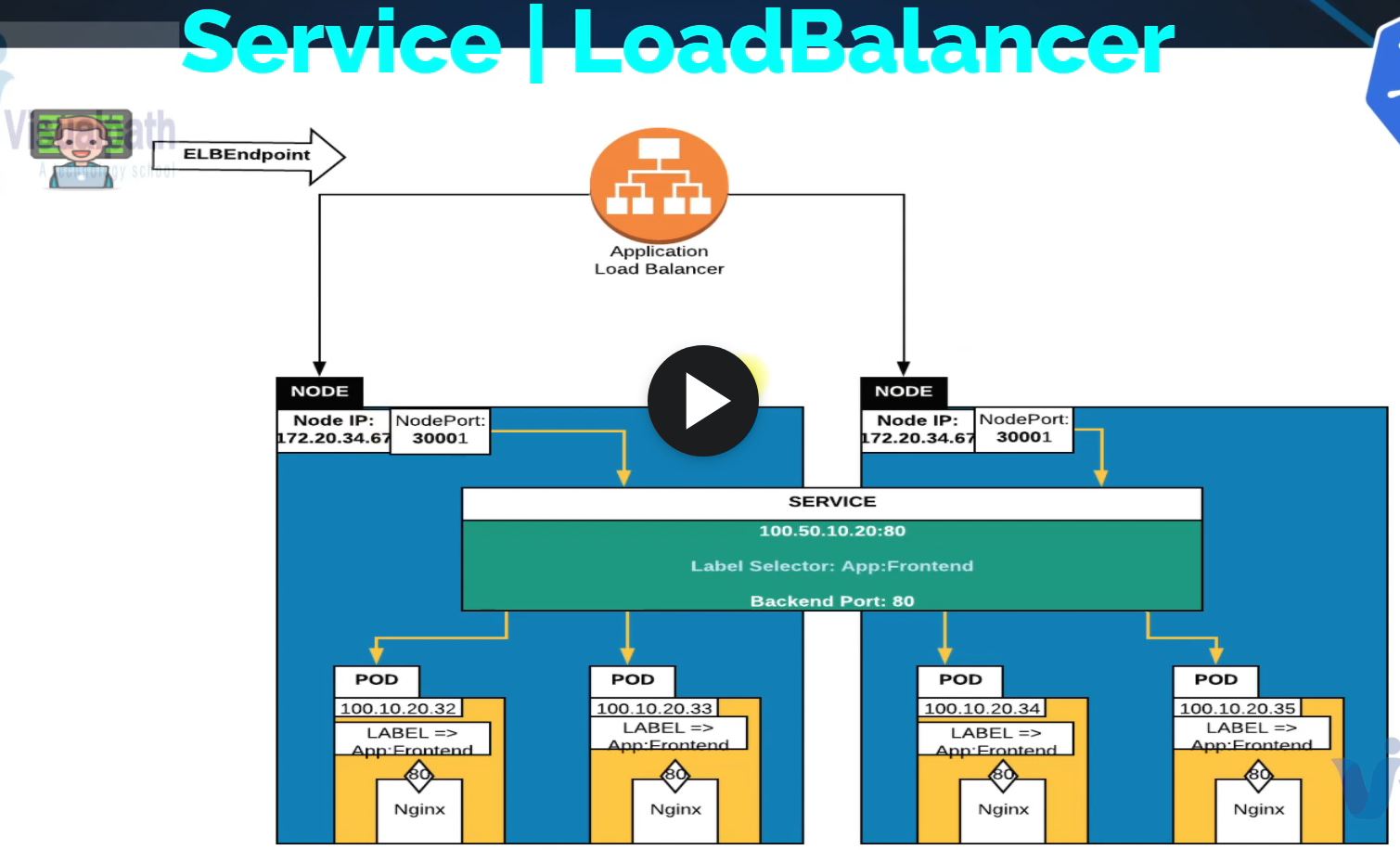
selector:

app: vproapp

type: LoadBalancer

:wq

1. kubectl create –f vproapp-loadbalancer.yml
2. kubectl get svc
3. Go to LoadBalancer under the AWS console, you will see the new created load balancer



1. Get the link and paste into your browser, you don’t need to mention any port. Your artifact should be up and running

**CLUSTER IP** (Type of Service)

apiVersion: v1

kind: Service

metadata:

name: app-service

spec:

ports:

- targetPort: 8080

port: 8080 (Internal frontend port)

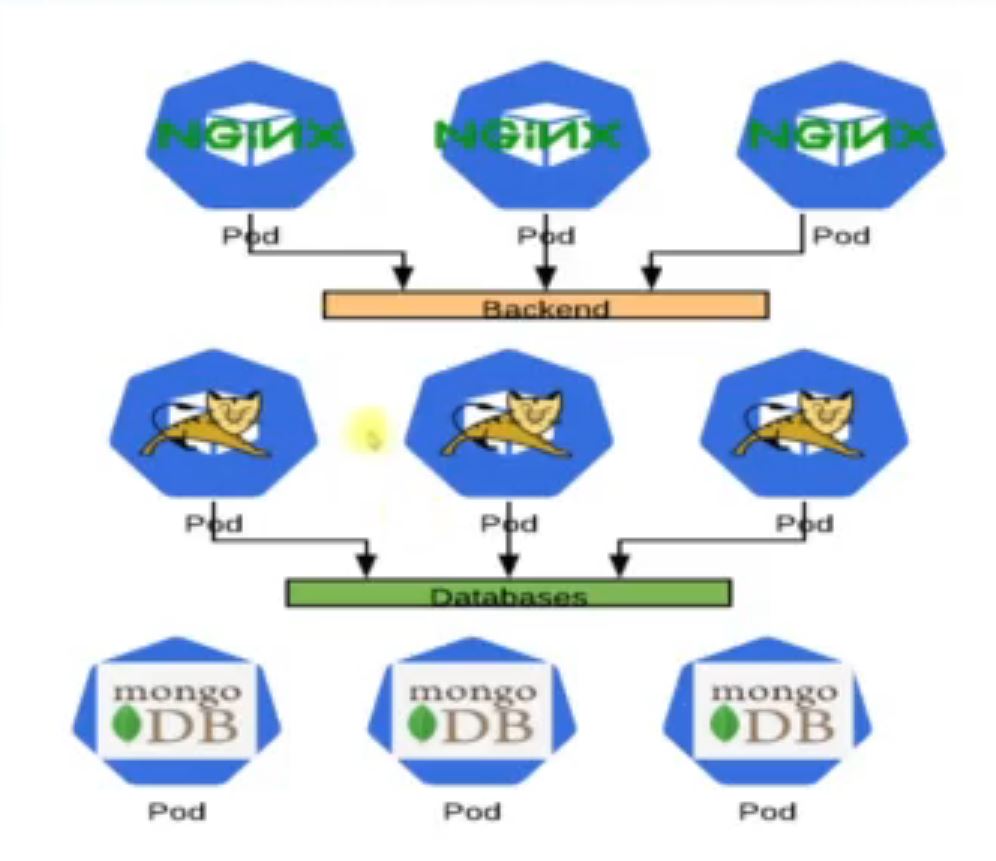
protocol: TCP

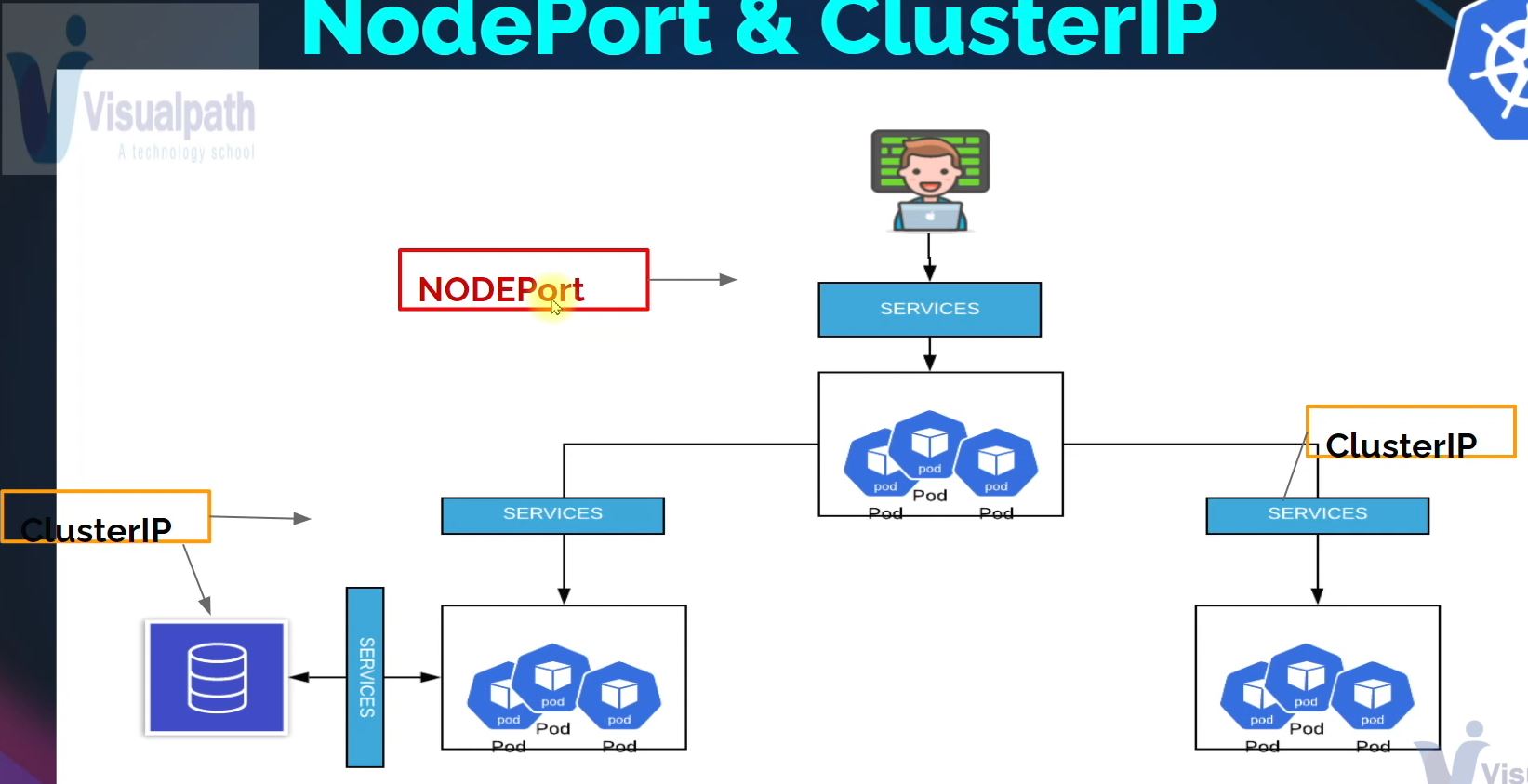
selector:

app: backend

type: ClusterIP

:wq





Clean Up

1. To do complete clean up:
2. kubectl delete service/helloworld-service
3. kubectl delete pod/vproapp
4. To confirm clean up
5. kubectl get all

**REPLICASET (Object in K8s)**

These maintains a Replica of your POD in a situation of failure. In Kubernetes, a ReplicaSet is an abstraction that defines a desired set of replicas of a Pod template. A ReplicaSet ensures that a specified number of replicas of a Pod are running at any given time, and it automatically replaces any Pods that fail or are deleted.

ReplicaSets are typically used to manage stateless applications, such as web servers or microservices, that can be scaled horizontally by adding or removing replicas.

Here are some key features of ReplicaSets:

1. Desired state: A ReplicaSet defines a desired state for a set of Pods, including the number of replicas and the Pod template to use.
2. Automatic scaling: ReplicaSets automatically scale the number of replicas up or down to match the desired state.
3. Self-healing: If a Pod fails or is deleted, the ReplicaSet automatically replaces it with a new Pod to maintain the desired state.
4. Rolling updates: ReplicaSets can be used to perform rolling updates of a set of Pods, ensuring that there is no downtime during the update process.

Overall, ReplicaSets provide a powerful abstraction for managing the deployment and scaling of stateless applications in Kubernetes. They ensure that a specified number of replicas are running at any given time, and they automatically replace any failed or deleted Pods to maintain the desired state.

HOW TO SETUP REPLICASETS:

1. SSH to your EC2 Kubes VM (Ubuntu created above)
2. Ubuntu user
3. vim replset.yaml

apiVersion: apps/v1

kind: ReplicaSet

metadata:

  name: frontend

  labels:

    app: guestbook

    tier: frontend

spec:

  # modify replicas according to your case

  replicas: 5

  selector:

    matchLabels:

      tier: frontend

  template:

    metadata:

      labels:

        tier: frontend

    spec:

      containers:

      - name: php-redis

        image: gcr.io/google\_samples/gb-frontend:v3

:wq

1. kubectl create –f replset.yaml
2. kubectl get rs
3. kubectl get pod
4. To scale up or Down
5. kubectl scale - -replicas=1 rs/frontend (These would reduce the replica to 1)
6. kubectl edit rs frontend (Adjust the replica to your desired size)
7. The two scaling methods in No 8 & 9 are not recommended in Production, you do it in the manifest and then apply the changes
8. For clean up
9. kubectl delete rs frontend

Deployment

In Kubernetes, a Deployment is an abstraction that provides declarative updates to a set of ReplicaSets, which in turn manage a set of Pods. A Deployment allows you to deploy a new version of your application, roll back to a previous version, or scale up or down the number of replicas, all with a simple declarative YAML configuration.

Deployments are typically used to manage stateful applications, such as databases or queues, that require more complex update strategies than stateless applications. Deployments provide a number of features to help manage updates, including rolling updates, canary deployments, and rollbacks.

Here are some key features of Deployments:

1. Declarative updates: Deployments allow you to declaratively specify the desired state of your application, and Kubernetes will manage the rollout and update process for you.
2. Rolling updates: Deployments support rolling updates, allowing you to update your application gradually, with a specified amount of downtime or traffic redirection.
3. Canary deployments: Deployments support canary deployments, allowing you to test a new version of your application on a small subset of the replicas before rolling it out to the entire deployment.
4. Rollbacks: Deployments support rollbacks, allowing you to quickly and easily revert to a previous version of your application in case of issues.

Overall, Deployments provide a powerful abstraction for managing the rollout and update of stateful applications in Kubernetes. They provide a number of features to help manage updates, including rolling updates, canary deployments, and rollbacks, making it easy to keep your application up-to-date and running smoothly.

**FORMAT FOR A DEPLOYMENT DEFINITION FILE**

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

STEPS:

1. Login to your kube vm on EC2
2. Ubuntu user
3. vim deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template: (Everything Under Template down to the end are infos about the POD)

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

:wq

1. kubectl apply –f deployment.yaml
2. kubectl get deploy
3. kubectl describe pod (insert a POD name)
4. To change the image file of a deployment, use the code below:
5. kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
6. kubectl get deploy
7. kubectl get pod
8. kubectl describe pod (put a pod name) (You will see the new updated Image name)
9. kubectl get rs (To view the new replicaset created)
10. To roll back to a previous deployment
11. kubectl rollout undo deployment/nginx-deployment
12. kubectl describe pod (POD name) | grep Image (You will see the former Image version)
13. You can see the roll back history log and do a particular rollback
14. kubectl rollout history deployment/(pod name)
15. kubectl rollout undo deployment/(pod name) - -to-revision=2
16. You can scale a deployment
17. kubectl scale deployment/(pod name) - -replicas=10
18. To clean up
19. kubectl delete deploy (name)

**VOLUMES:** (Objects in K8s)

In Kubernetes, a Volume is a way to persist data beyond the lifetime of a Pod. Volumes are used to store and share data between containers in a Pod, or between a container in a Pod and the host machine.

Volumes are similar to Docker volumes or network file systems, but they are designed to work seamlessly with Kubernetes Pods. A Volume can be used to store any type of data that needs to be preserved beyond the lifetime of a Pod, such as a database, configuration files, or logs.

Here are some key features of Volumes:

1. Persistence: Volumes provide a way to store data that persists beyond the lifetime of a Pod, even if the Pod is deleted or recreated.
2. Sharing: Volumes can be shared between containers in a Pod, or between a container in a Pod and the host machine.
3. Flexibility: Volumes can be backed by a variety of storage types, including local disk, network file systems, cloud storage, and more.
4. Security: Volumes can be encrypted and secured using Kubernetes secrets or other encryption methods.

Overall, Volumes provide a powerful way to store and share data in Kubernetes, making it easy to persist data beyond the lifetime of a Pod and share data between containers and the host machine.

**TYPES OF K8s VOLUMES:**

In Kubernetes, there are several types of Volumes that can be used to store and share data between containers in a Pod or between a container in a Pod and the host machine. Here are some of the most common types of Kubernetes Volumes:

1. EmptyDir: This volume type is created when a Pod is scheduled on a node and is deleted when the Pod is terminated. It can be used to store temporary data or to share data between containers in a Pod.
2. HostPath: This volume type mounts a file or directory from the host machine into a Pod. It can be used to share data between a container in a Pod and the host machine.
3. ConfigMap: This volume type provides a way to store and share configuration data as key-value pairs between containers in a Pod.
4. Secret: This volume type provides a way to store and share sensitive data, such as passwords or API keys, between containers in a Pod.
5. PersistentVolumeClaim (PVC): This volume type provides a way to request a specific amount of storage from a StorageClass defined in the Kubernetes cluster. PVCs can be used to store data that needs to persist beyond the lifetime of a Pod.
6. NFS: This volume type uses the Network File System (NFS) to mount a file system from a remote server onto a Pod.
7. StorageOS: This volume type uses the StorageOS software-defined storage platform to provide scalable, persistent storage for Kubernetes clusters.

These are just some examples of the types of Volumes available in Kubernetes. Each Volume type has its own features and benefits, so it's important to choose the right Volume type for your specific use case.

**STEPS TO WORK WITH K8S VOLUMES: (Please note this is just for test environment)**

1. vim mysqlpod.yaml

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: dbpd

**spec**:

**containers**:

- **image**: mysql:5.7

**name**: mysql

**volumeMounts**:

- **mountPath**: /var/lib/mysql

**name**: dbvolume

**volumes**:

- **name**: dbvol

**hostPath**:

*# directory location on host*

**path**: /data

*# this field is optional*

**type**: DirectoryOrCreate

:wq

1. kubectl apply –f mysqlpod.yaml
2. kubectl get pod
3. kubectl describe pod dbpod

**CONFIG MAP:** (Objects in K8s)

I will explain practically in these section Environment Variables, configuration files and Commanline Arguments. How to inject variables and configuration in a POD:

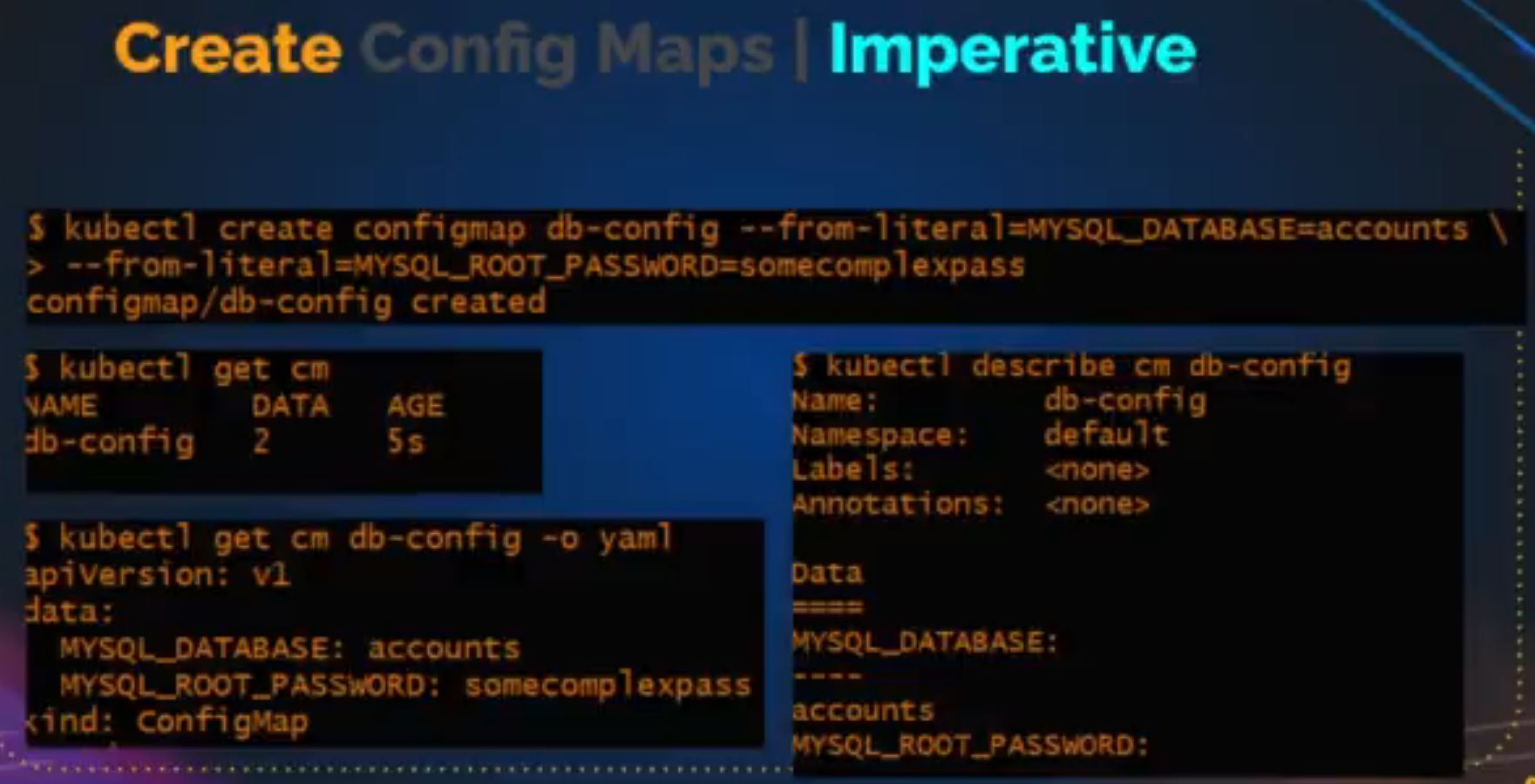
In Kubernetes, ConfigMaps are used to store configuration data that can be used by a containerized application running in a Pod. ConfigMaps provide a way to decouple configuration from the container image, making it easier to manage and update configuration data without having to rebuild the container image.

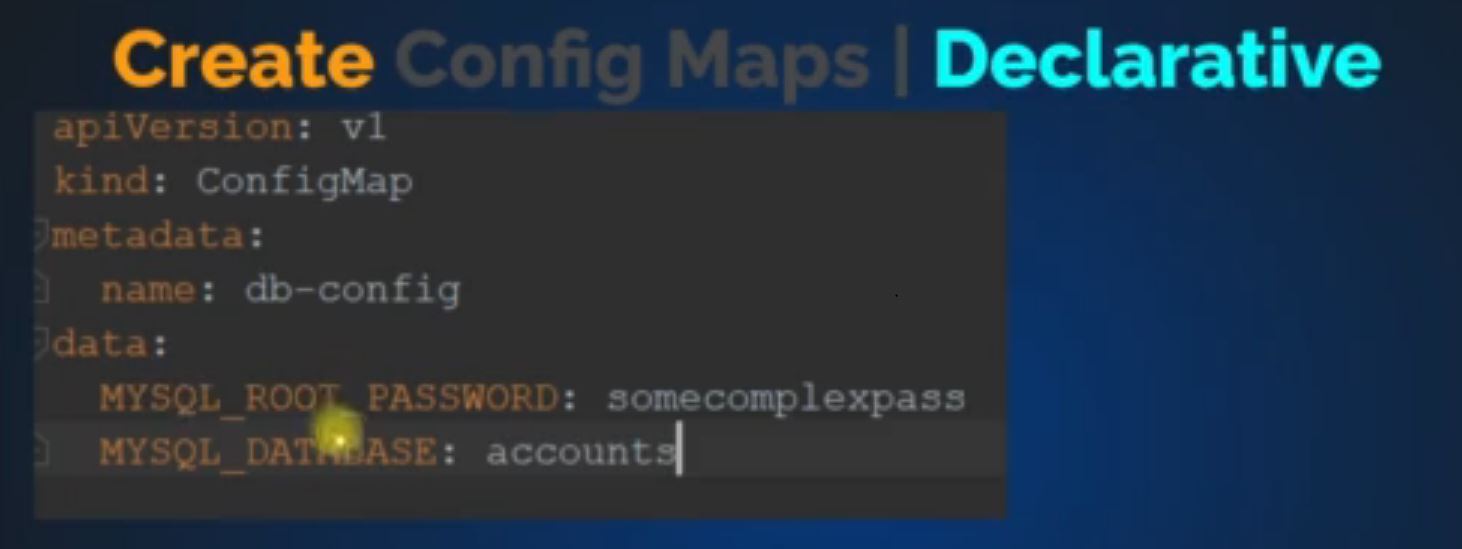
ConfigMaps can be created using YAML or JSON files, or they can be created from environment variables or command-line arguments. The data stored in a ConfigMap can be accessed by a container in a Pod using environment variables or as files mounted in a Volume.

Here are some examples of the types of configuration data that can be stored in a ConfigMap:

1. Environment variables: You can store environment variables in a ConfigMap and then inject them into a container running in a Pod. This makes it easy to change configuration values without having to rebuild the container image.
2. Command-line arguments: You can store command-line arguments in a ConfigMap and then use them to configure a container running in a Pod.
3. Configuration files: You can store configuration files, such as application.properties or config.ini, in a ConfigMap and then mount them as a Volume in a container running in a Pod.
4. Secrets: You can store sensitive data, such as passwords or API keys, in a ConfigMap by encoding them using base64 encoding. While this is not considered secure, it provides an easy way to manage sensitive data in a ConfigMap.

ConfigMaps can be created using the **kubectl create configmap** command or by using YAML or JSON files. Once a ConfigMap is created, it can be used to configure one or more containers running in a Pod.



****

**STEPS TO WORK WITH CONFIG MAPS:**

1. vim samplecm.yaml

**apiVersion**: v1

**kind**: ConfigMap

**metadata**:

**name**: game-demo

**data**:

*# property-like keys; each key maps to a simple value*

**player\_initial\_lives**: "3"

**ui\_properties\_file\_name**: "user-interface.properties"

*# file-like keys*

**game.properties**: |

*enemy.types=aliens,monsters*

*player.maximum-lives=5*

**user-interface.properties**: |

*color.good=purple*

*color.bad=yellow*

*allow.textmode=true*

:wq

1. kubectl apply –f samplecm.yaml
2. kubectl get cm
3. kubectl get cm (name) -o yaml
4. vim readcmpod.yaml

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: configmap-demo-pod

**spec**:

**containers**:

- **name**: demo

**image**: alpine

**command**: ["sleep", "3600"]

**env**:

*# Define the environment variable*

- **name**: PLAYER\_INITIAL\_LIVES *# Notice that the case is different here*

*# from the key name in the ConfigMap.*

**valueFrom**:

**configMapKeyRef**:

**name**: game-demo *# The ConfigMap this value comes from.*

**key**: player\_initial\_lives *# The key to fetch.*

- **name**: UI\_PROPERTIES\_FILE\_NAME

**valueFrom**:

**configMapKeyRef**:

**name**: game-demo

**key**: ui\_properties\_file\_name

**volumeMounts**:

- **name**: config

**mountPath**: "/config"

**readOnly**: **true**

**volumes**:

*# You set volumes at the Pod level, then mount them into containers inside that Pod*

- **name**: config

**configMap**:

*# Provide the name of the ConfigMap you want to mount.*

**name**: game-demo

*# An array of keys from the ConfigMap to create as files*

**items**:

- **key**: "game.properties"

**path**: "game.properties"

- **key**: "user-interface.properties"

**path**: "user-interface.properties"

**COMMAND & ARGUMENTS**

