Kubernetes (K8s)

What is Kubernetes?

- **Kubernetes is the most popular container orchestration tool.**
- Manages multiple Docker nodes for high availability.

Why Kubernetes is Famous

- Prevents single point of failure by clustering Docker nodes.
- Distributes containers across multiple nodes for fault tolerance.
- Automatically restarts containers on healthy nodes if a node fails.
- Offers robust container orchestration for production environments.

Container Orchestration Tools

- **Docker Swarm:** Directly from Docker.
- **Kubernetes:** The most famous.
- **Mesosphere Marathon:** From Apache.
- **Cloud-Based Solutions:**
    - AWS ECS and EKS (Elastic Kubernetes Service).
    - Azure Container Service.
    - Google Container Engine.
- **In-House Solutions:** Custom orchestration platforms.

The Origin and Popularity of Kubernetes

- **Developed by Google:**
    - Based on their internal tool "Borg" for managing Linux containers.
    - Kubernetes introduced as an open-source project in mid-2014.
    - Stable version released in mid-2015.
    - Managed by CNCF (Cloud Native Computing Foundation).

Key Milestones

- **2016:** Tools like Kops and Minikube developed for Kubernetes.
- **2016:** Pokemon Go case study boosted confidence in Kubernetes for production.
- **2017:**
    - Enterprise adoption increased.
    - Google, IBM, and GitHub adopted Kubernetes.
    - Oracle joined CNCF.

Kubernetes Use at Google

- Google runs all its services, like Gmail, on Linux containers.
- **In 2014:** Google launched over 2 billion containers weekly.
- Kubernetes is a mature platform used by Google for decades.

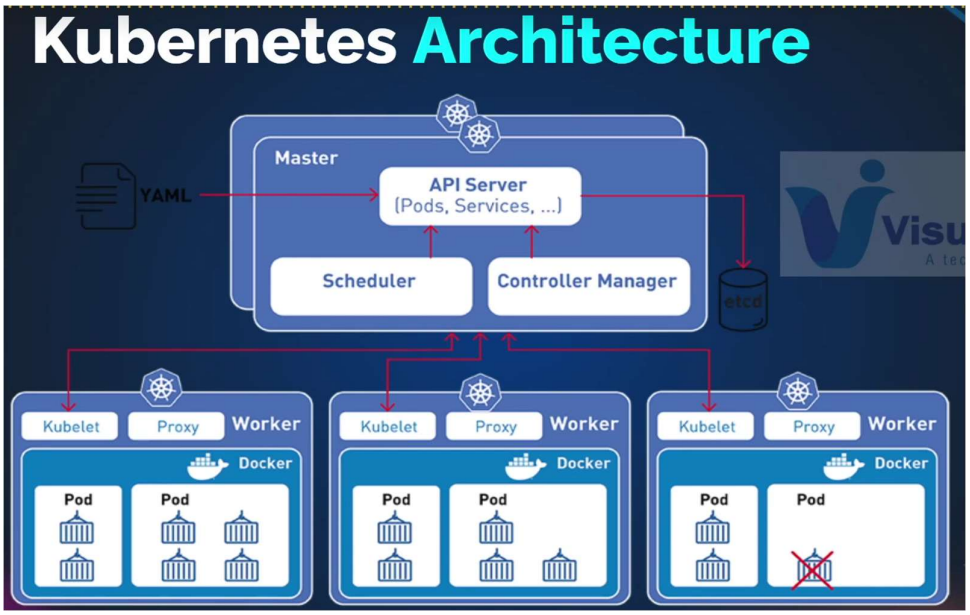
Kubernetes and Docker Engine

- **Kubernetes is not a replacement for Docker Engine.**

- **Kubernetes manages clusters of Docker engines**.
- Kubernetes can also manage clusters of other container runtime environments like Rocket.

Amazing Features of Kubernetes

1. **Service Discovery and Load Balancing**
    - Kubernetes provides **automatic service discovery and load balancing**.
    - Containers, referred to as "pods" in Kubernetes, are automatically discovered by the load balancer.
    - The load balancer updates automatically when new pods are created.
2. **Storage Orchestration**
    - Kubernetes integrates with various storage solutions, including:
        - SAN (Storage Area Network)
        - NAS (Network Attached Storage)
        - EBS (Elastic Block Store) volumes
        - Ceph storage
    - This wide range of integration options boosts confidence in running stateful containers.
3. **Automated Rollback**
    - **Easy to roll out new image versions**.
    - **Simple rollback** if a new version is not working correctly.
    - The rollback process is faster than in other environments like AWS Beanstalk.
4. **Automatic Bin Packing**
    - **Optimizes resource utilization** by placing containers on the most appropriate nodes.
    - Ensures that containers get the required resources based on their specifications.
5. **Self-Healing**
        - **Orchestration tools** ensure containers are managed efficiently.
        - If a node goes down, Kubernetes can **resurrect containers on a live node**.
        - This process is faster than traditional auto-scaling groups.
        - Monitors container health and replaces failed containers automatically.
6. **Configuration Management**
    - Manage configurations through **variables, volumes, and secrets**.
    - Secrets are encoded values, providing secure management of sensitive information.



Overview of Kubernetes Architecture

- Kubernetes architecture comprises various services that collectively form the complete Kubernetes cluster.
- The architecture is divided into two main components:
  - **Master Node (Control Plane)**
  - **Worker Node**

Master Node (Control Plane)

- **Master Node** manages the worker nodes and orchestrates the cluster.
- Users do not directly log into the worker nodes or the master node. Instead, they interact with the master node via a client.
- The master node (also known as the control plane) handles the deployment and management of containers across the cluster.

Key Services in the Master Node (Control Plane)

1. **API Server**
   - Serves as the frontend for the Kubernetes control plane.
   - Receives REST requests for modifications and updates the status of the cluster.
2. **Scheduler**
   - Assigns work to the worker nodes.
   - Decides which pod runs on which node based on resource availability.
3. **Controller Manager**
   - Runs controller processes to manage the state of the cluster.
   - Ensures that the cluster matches the desired state specified by the user.
4. **etcd**
   - A distributed key-value store used for storing all cluster data.
   - Provides consistent and reliable storage for configuration data and state information.

Worker Node

- **Worker Nodes** are where the actual application containers are run, powered by Docker Engine or other container runtimes.
- The master node assigns tasks to worker nodes, but users do not interact with worker nodes directly.

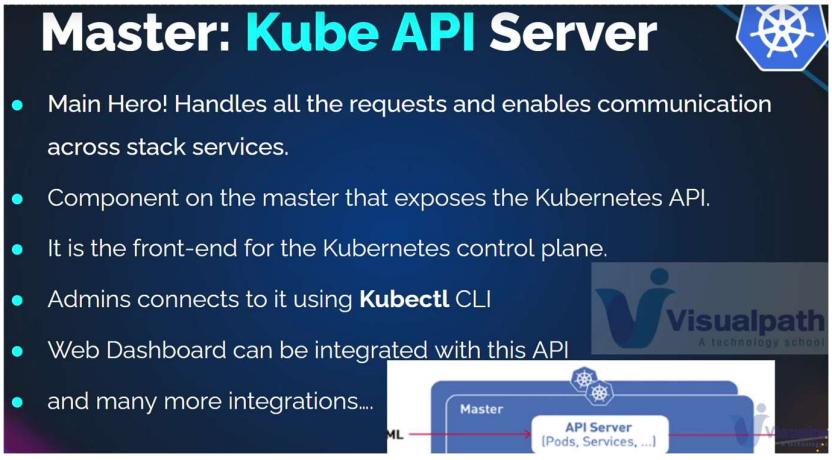
Key Services in the Worker Node

- **Kubelet**
  - An agent that runs on each worker node.
  - Ensures that containers are running in a pod.
  - Communicates with the API server and executes the instructions provided by the control plane.
- **Kube-proxy**
  - Maintains network rules on nodes.
  - Enables network communication to the pods from network sessions inside or outside of the cluster.
- **Docker Engine**
  - The container runtime that runs the actual application containers.

Control Plane (Master Node)

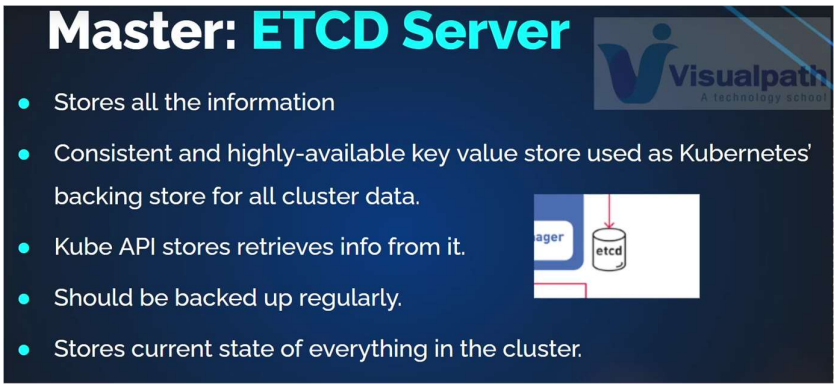
1. Kube API Server

- Handles all incoming and outgoing communication.
- Exposes Kubernetes API for integration with third-party tools.
- Frontend for the control plane, accessible via **kubectl** CLI.
- Supports integrations like monitoring agents, logging agents, and web dashboards.



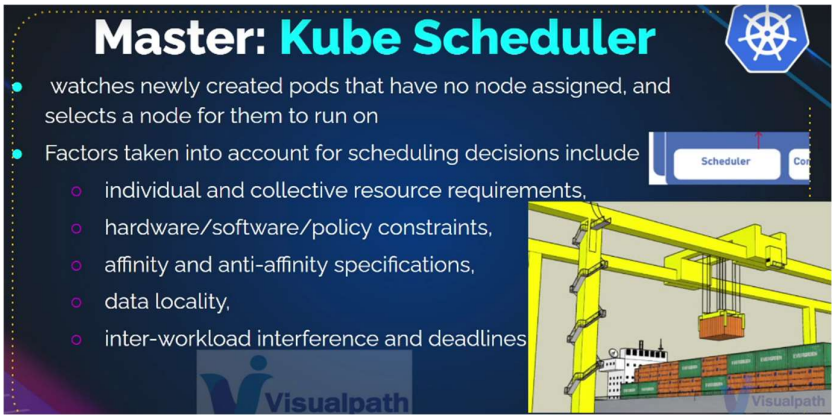
2. etcd

- Key-value store for cluster data.
- Stores runtime information and the current state of the cluster.
- Regular backups are crucial to prevent data loss.



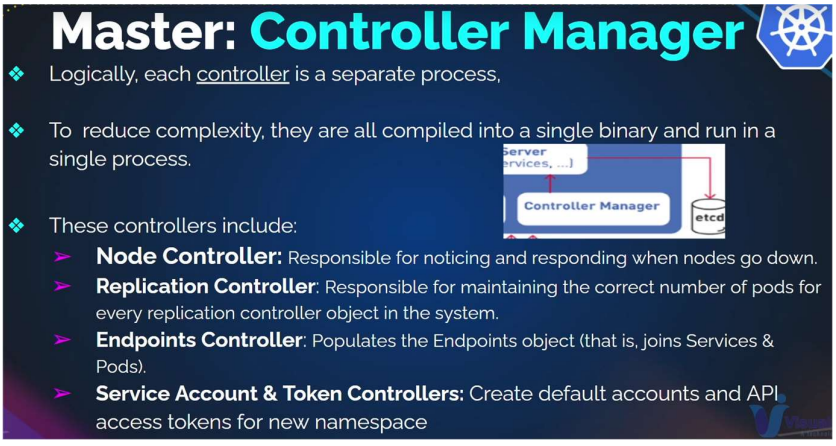
3. Scheduler

- Schedules containers on the appropriate worker nodes.
- Considers factors like resource requirements, hardware/software specifications, and affinity/anti-affinity rules.



4. Controller Manager

- Group of controllers managing different cluster aspects:
  - **Node Controller:** Monitors and takes action if worker nodes go down.
  - **Replication Controller:** Monitors and auto-heals pods (containers).
  - **Endpoint Controller:** Manages endpoint objects.
  - **Service Account & Token Controller:** Manages authentication and authorization.



Worker Node

1. Kubelet

- Agent running on each worker node.
- Listens to master node commands and manages container execution.

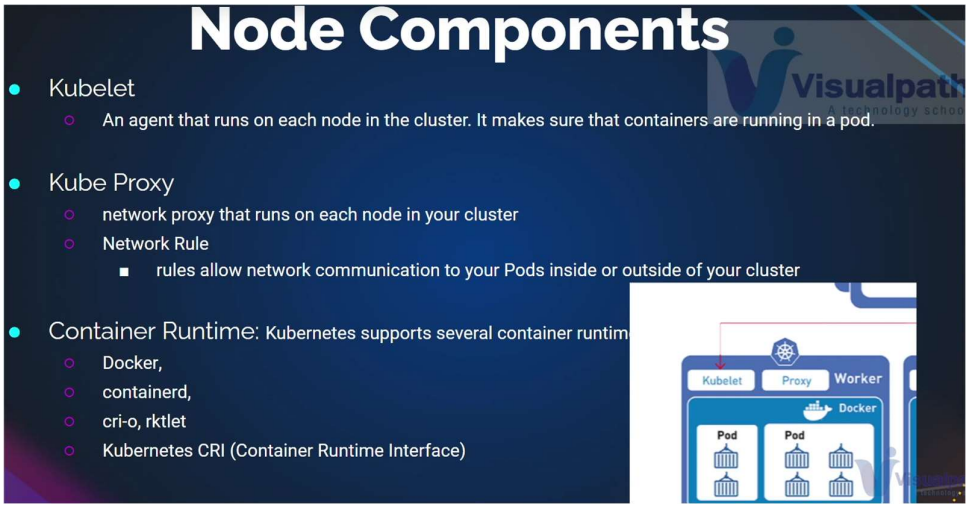
- Fetches images and runs containers, handling commands like **docker run**.

2. Kube-proxy

- Network proxy running on each node.
- Manages network rules similar to security groups.
- Can expose pods to the outside world and set network policies.

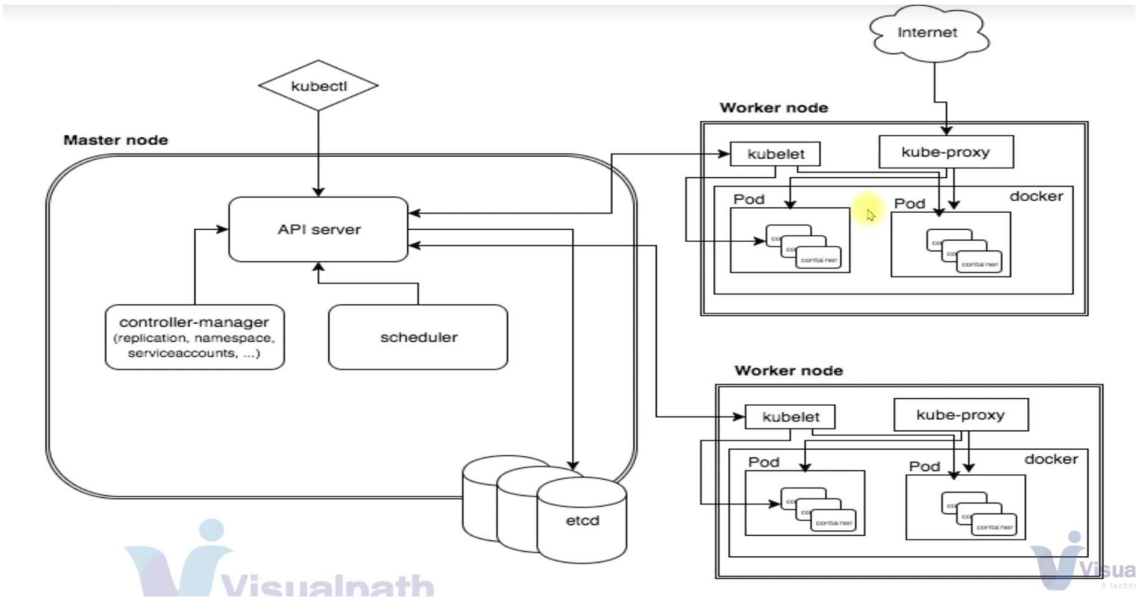
3. Container Runtime Environment

- Supports various container runtimes: Docker, containerd, Rocket, Kubernetes CRI.
- Flexible compared to Docker Swarm, which only supports Docker Engine.



Additional Components

- **Add-ons:**
    - DNS, web UI, resource monitoring, and cluster-level logging.
    - Often provided by specialized third-party vendors for enhanced functionality.
- **Control Plane:**
    - API Server: Communication.
    - etcd: Data storage.
    - Scheduler: Node selection for containers.
    - Controller Manager: Monitoring and management.
- **Worker Node:**
    - Kubelet: Executes container commands.
    - Kube-proxy: Manages network rules.
    - Container Runtime: Runs containers in various environments.



What is a Pod?

- **Pod and Container Relationship:**
  - A Pod is to a container what a VM is to a process.
  - Provides resources (network, RAM, CPU, storage) to the container.
  - Container runs inside the Pod, similar to a process running inside a VM.
  - No virtualization, only isolation.
- **Purpose of Pods in Kubernetes:**
  - Abstraction layer to support different container runtime environments (Docker, Rocket, CRI).
  - Standard set of commands and configurations regardless of underlying technology.

How Pods Work

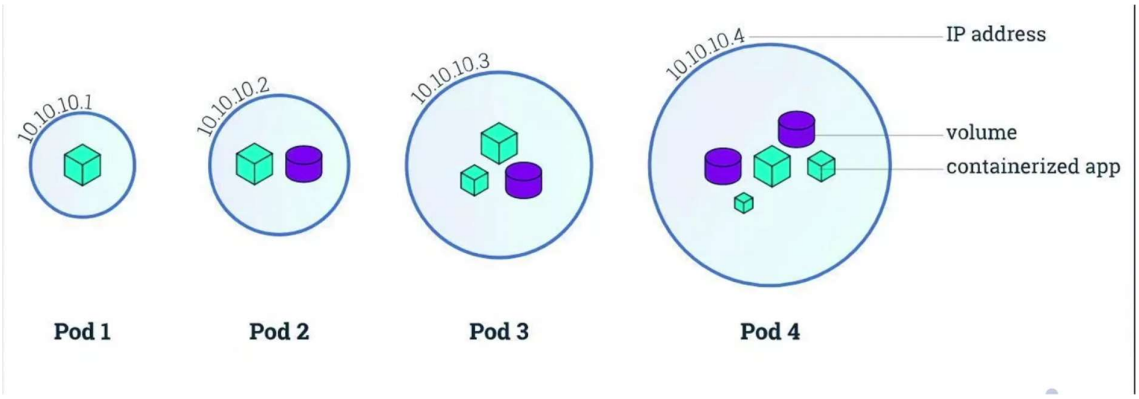
- **Pod Configuration:**
  - Pods provide IP addresses and port mappings to containers.
  - Example: Tomcat running in a Pod with IP and port accessible externally.
- **Pod Composition:**
  - Can contain one or multiple containers.
  - Resources (e.g., volumes) can be shared among containers in a Pod.
- **Pod Examples:**
  - **Single Container:** One container in a Pod.
  - **Container with Volume:** Container with an attached volume.
  - **Multiple Containers with Volume:** Two containers sharing a volume.
- **Container Roles in Pods:**
  - **Main Container:** Primary container in the Pod.
  - **Helper Containers:**
    - **Sidecar Container:** Assists the main container (e.g., logging, monitoring).
    - **Init Container:** Short-lived, performs initialization tasks before main container starts.
- **Best Practices:**
  - Typically, only one main container per Pod.
  - Helper containers support the main container but are not independent.
- **Pod Distribution and Interaction**
- **Pods Across Nodes:**
  - Pods can be distributed across multiple worker nodes.
  - Example: Pod One (Tomcat) in Node One, Pod Six (MySQL) in Node Three.
- **Pod Communication:**

    - **Overlay Network:**
        - Similar to a VPC in AWS.
        - Connects all nodes in a virtual network.
        - Each node has a subnet, a private network.
- **Network Components:**
    - **Bridge Zero:** Acts like a switch within a node, facilitating communication between Pods in the same node.
    - **WG Zero:** Acts like a router, forwarding requests between nodes.
- **Network Flow:**
    - Bridge Zero forwards intra-node requests.
    - WG Zero routes inter-node requests by IP address, connecting to the appropriate node's bridge zero.

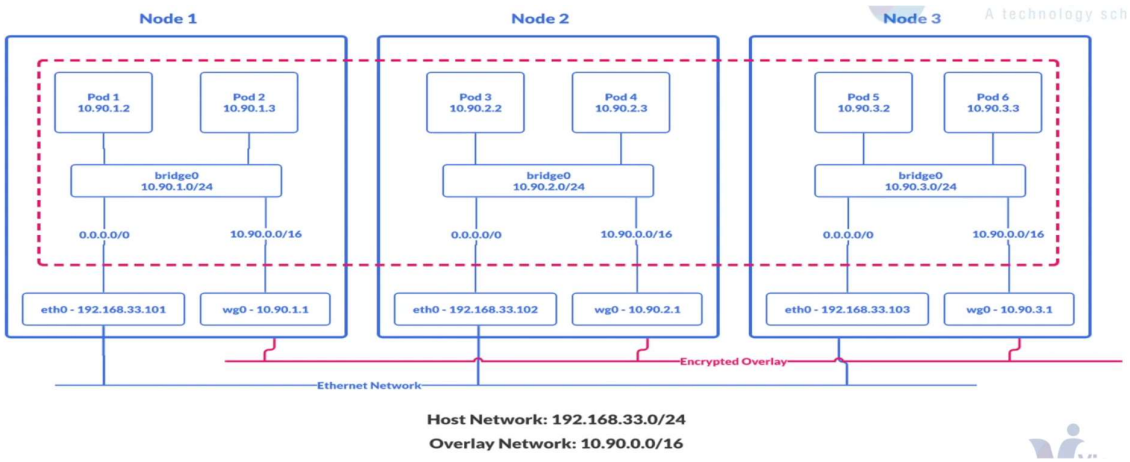
Summary of Key Concepts

- **Pod:** Provides an abstraction layer and resources for containers.
- **Container Runtime:** Supports multiple environments (Docker, Rocket, CRI).
- **Network:** Uses an overlay network for communication across nodes.
- **Components:**
    - Bridge Zero: Switch within a node.
    - WG Zero: Router for inter-node communication.

PODS



Overlay Network



Methods to Set Up a Kubernetes Cluster

1. **Manual Setup:**
    - The hardest way to set up a Kubernetes cluster.
    - Recommended only for understanding the underlying components and processes.
2. **MiniKube:**
    - Best for testing and learning on a local machine with a single node.
    - **Purpose:** Testing and learning.
    - **Setup:**
        - Launches a single-node Kubernetes cluster on your computer.
        - Uses VirtualBox to create a VM where both master and worker node components run.
    - **Limitations:** Not suitable for production environments.
3. **Kubeadm:**
    - Ideal for creating multi-node clusters for production on various platforms.
    - **Purpose:** Production-ready, multi-node Kubernetes clusters.
    - **Setup:**
        - Facilitates the creation of clusters with multiple nodes.
        - Requires logging into the master node and running specific commands.
        - Requires running commands on each worker node to connect them to the master node.
    - **Flexibility:** Can be used on physical machines, virtual machines, or cloud platforms.
4. **Kops:**
    - **Purpose:** Stable, production-ready Kubernetes clusters.
    - **Setup:**
        - Initially designed for AWS but now supports Google Cloud, Digital Ocean, and OpenStack.
    - **Advantages:**
        - Streamlines the setup process.
        - Ensures stability and scalability for production environments.
    - Provides a stable and scalable way to set up production clusters, especially useful for cloud environments.

Minikube Setup

1. **Install Chocolatey:**
    - Open PowerShell as an administrator.
    - Install Chocolatey by running the 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://community.chocolatey.org/install.ps1'))**
2. **Install Minikube and Kubernetes CLI:**
    - Using Chocolatey, install Minikube and kubectl by running:
    - **choco install minikube kubernetes-cli -y**
    - Restart PowerShell to apply changes.
3. **Start Minikube:**
    - Open a new PowerShell window and start Minikube:
    - **minikube start**
    - This command will create a VM on Oracle VM VirtualBox and launch a single-node Kubernetes cluster.
4. **Verify Minikube Setup:**
    - Check the status of the Minikube cluster:
    - **kubectl get nodes**

https://minikube.sigs.k8s.io/docs/start/

5. **Test Minikube Deployment:**
    - Create a deployment: **kubectl create deployment hello-minikube --image=kicbase/echo-server:1.0**
    - Verify the deployment and pod:
    - **kubectl get pod**
    - **kubectl get deploy**
    - Expose the deployment as a service: **kubectl expose deployment hello-minikube --type=NodePort --port=8080**
    - Get the service URL:
    - **kubectl get services hello-minikube**
6. **Cleanup Minikube Cluster:**
    - Delete the deployment and service:
    - **kubectl delete svc hello-minikube**
    - **kubectl delete deploy hello-minikube**
    - Stop and delete the Minikube cluster:
    - **minikube stop**
    - **minikube delete**

Setup with Kops

- Domain for Kuberenetes DNS records
  e.g. groophy.in from GoDaddy
- Create a linux VM and setup kops, kubectl, ssh keys, awscli
- Login to AWS account and setup
  S3 bucket, IAM User for AWSCli, Route 53 Hosted Zone

Initiate an ec2 instance of kops of ubuntu , kops is not part of Kubernetes cluster , this is just an instance to launch the Kubernetes cluster

Prerequisites:

1. **Domain:**
    - Purchase a domain to manage Kubernetes DNS records. (e.g., from GoDaddy)
2. **Linux VM:**
    - Set up a Linux VM using Vagrant or EC2 instance on AWS. This VM will be used to set up Kops, kubectl, SSH keys, and AWS CLI. It will not be part of the Kubernetes cluster.
3. **AWS S3 Bucket:**
    - Create an S3 bucket to store cluster state.
4. **AWS IAM User:**
    - Create an IAM user with Administrator Access.
5. **AWS Route 53 Hosted Zone:**
    - Create a Route 53 hosted zone for the subdomain.

Steps to Set Up Kops:

1. **Set Up EC2 Instance:**
    - Launch an EC2 instance with Ubuntu AMI.
    - Configure network settings and create a security group allowing port 22.
    - Create a key pair for SSH access.
2. **Install Required Tools:**
    - SSH into the EC2 instance and update packages:
    - sudo apt update
    - Install AWS CLI
    - sudo apt install awscli
    - Install kubectl
    - curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl" chmod +x kubectl sudo mv kubectl /usr/local/bin/
    - Install Kops:
    - curl -LO "https://github.com/kubernetes/kops/releases/download/v1.26.4/kops-linux-amd64" chmod +x kops-linux-amd64 sudo mv kops-linux-amd64 /usr/local/bin/kops
3. **Configure AWS CLI:**

- Run **aws configure** and provide the access key, secret key, region (e.g., **us-east-2**), and output format (**json**).

4. **Generate SSH Keys:**
    - Generate SSH keys for Kops:
    - ssh-keygen
5. **Create S3 Bucket:**
    - Create an S3 bucket for Kops state
    - aws s3api create-bucket --bucket <your-unique-bucket-name> --region us-east-2 --create-bucket-configuration LocationConstraint=us-east-2
6. **Set Up Route 53 Hosted Zone:**
    - Create a hosted zone in Route 53 for your subdomain.
7. **Configure DNS in Domain Registrar:**
    - Add NS records in your domain registrar to point to the Route 53 name servers.
8. **Create Kubernetes Cluster with Kops:**
    - Create the cluster configuration:

```
ubuntu@ip-172-31-12-44: ~
ubuntu@ip 172 31 12 44:~$ kops create cluster    name=kubevpro.groophy.in \
> --state=s3://vprofile-kop-states --zones=us-east-2a,us-east-2b \
> --node-count=2 --node-size=t3.small --master-size=t3.medium --dns-zone=kubevpro.groophy.in \
> --node-volume-size=8 --master-volume-size=8
```

    - kops create cluster --name=<subdomain.domain.com> --state=s3://<your-unique-bucket-name> --zones=us-east-2a,us-east-2b --node-count=2 --node-size=t3.small --master-size=t3.medium --node-volume-size=8 --master-volume-size=8 --dns-zone=<subdomain.domain.com>
    - Apply the configuration to create the cluster:
    - kops update cluster --name <subdomain.domain.com> --state=s3://<your-unique-bucket-name> --yes
9. **Validate Cluster Setup:**
    - Validate the cluster:

```
ubuntu@ip 172 31 12 44:~$ kops validate cluster    state=s3://vprofile kop states
```

    - kops validate cluster --state=s3://<your-unique-bucket-name>
    - Verify nodes with kubectl:
    - kubectl get nodes
10. **Cleanup Kops Cluster:**
    - Delete the cluster:
    - kops delete cluster –name=<subdomain.domain.com> --state=s3://<your-unique-bucket-name> --yes
11. **Power Off EC2 Instance:**

- Power off the EC2 instance used for Kops setup:
- sudo poweroff

By following these steps, you can set up a Kubernetes cluster for both testing (with Minikube) and production (with Kops). Make sure to manage your AWS resources carefully to avoid unnecessary charges.

Understanding Kubernetes Objects

In Kubernetes, objects are the persistent entities in the system. They represent the state of your cluster, including the applications you want to run and the policies around them. Here are some key Kubernetes objects that we will cover in this course:

1. **Pod:**
    - The smallest and simplest Kubernetes object.
    - Represents a single instance of a running process in your cluster.
    - A pod can contain one or more containers.
    - In Kubernetes, you do not manage containers directly; instead, you manage pods, which in turn manage the containers.
2. **Service:**
    - Provides a stable endpoint (IP address and port) for a set of pods.
    - Acts as a load balancer within the cluster.
    - Useful for exposing your applications to the external world or for internal communication within the cluster.
3. **ReplicaSet:**
    - Ensures a specified number of pod replicas are running at any given time.
    - Useful for maintaining high availability and fault tolerance.
4. **Deployment:**
    - Provides declarative updates for pods and ReplicaSets.
    - Allows you to describe an application's lifecycle, such as which images to use for the app, the number of pod replicas, and update policies.
    - One of the most commonly used objects for deploying applications in a Kubernetes cluster.
5. **ConfigMap:**
    - Used to store configuration data in key-value pairs.
    - Decouples configuration artifacts from image content to keep containerized applications portable.
6. **Secret:**
    - Similar to ConfigMap but is intended to hold sensitive information, such as passwords, OAuth tokens, and SSH keys.
    - Provides a way to pass confidential data to your applications.
7. **Volume:**
    - Provides a way to attach storage to your pods.
    - Different types of volumes are supported, such as persistent volumes, emptyDir, hostPath, and more.

Kube Config File

Connecting kubectl to Kubernetes Cluster

When you first run kubectl commands, it might be unclear how kubectl connects to your Kubernetes cluster. It needs to know the location of your master node and how to authenticate and interact with the cluster. This information is stored in the **kubeconfig** file.

What is the Kubeconfig File?

The kubeconfig file contains:

- **Cluster information**: The location and credentials of your Kubernetes cluster(s).
- **User information**: Credentials and authentication methods.
- **Namespace**: The default namespace for kubectl commands.
- **Contexts**: Combine cluster, user, and namespace information to specify which cluster kubectl should interact with by default.

Structure of Kubeconfig File

Here's a sample kubeconfig file structure:

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: <base64 encoded cert>
    server: https://<api-server-endpoint>
  name: my-cluster
users:
- name: my-user
  user:
    client-certificate-data: <base64 encoded cert>
    client-key-data: <base64 encoded key>
contexts:
- context:
    cluster: my-cluster
    user: my-user
    namespace: my-namespace
  name: my-context
```

```
current-context: my-context
```

Explanation of Kubeconfig File Sections

- **clusters**: Contains information about the cluster(s), such as the API server endpoint and certificates.
- **users**: Specifies user credentials and certificates for authentication.
- **contexts**: Binds clusters and users together, specifying which user to use for which cluster.
- **current-context**: The default context used by kubectl commands.

Now copy the kube/config file from ur instance machine and copy it in ur local machine in .kube/config location

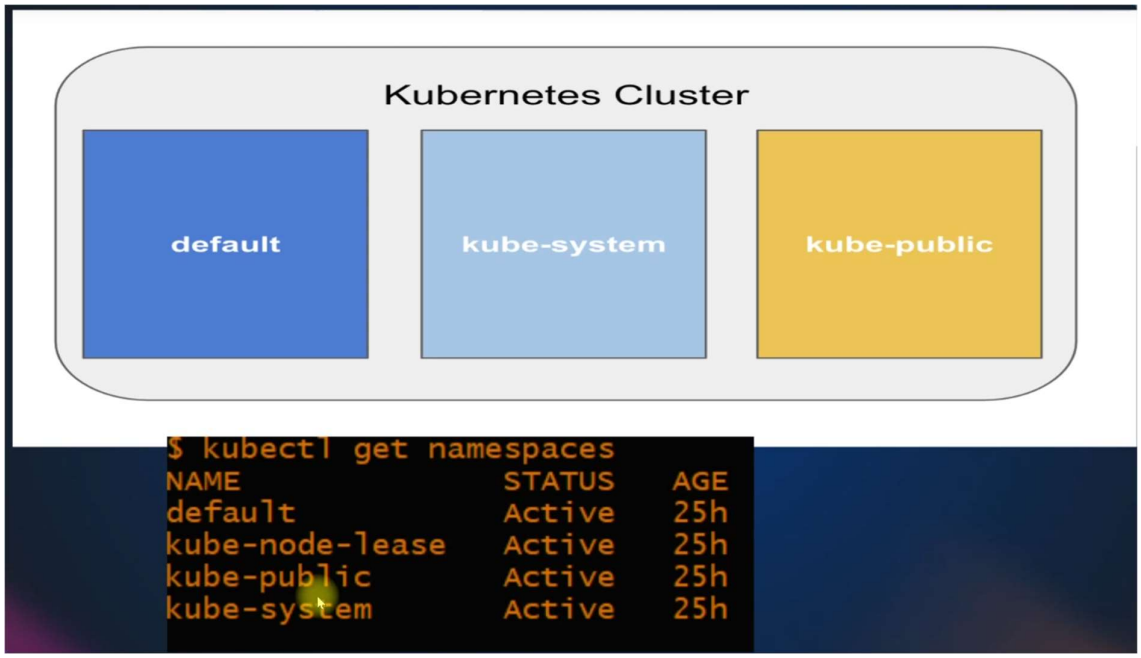
Now install kubectl in windows using chocolatey , verify the installation

Read the config file documentation on k8s docs

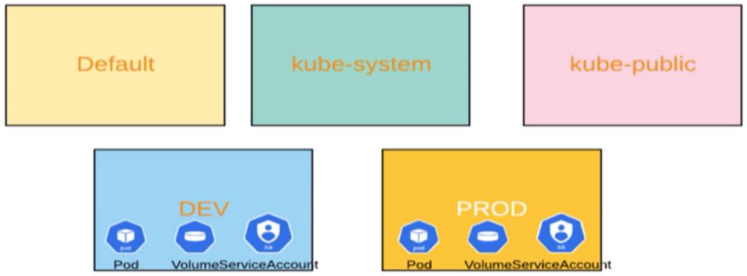
Summary

- **Kubeconfig File**: Essential for kubectl to interact with Kubernetes clusters.
- **Multiple Clusters and Users**: Kubeconfig can manage multiple clusters and users.
- **Contexts**: Specify which cluster and user to use by default.
- **Documentation**: Use Kubernetes documentation for detailed guidance.

Namespaces



For example, for your development environment, you can create a different namespace for production, you can create a different namespace and you can isolate your environments like this. You can also create different namespaces for different projects.



Kubernetes Namespaces

Introduction to Namespaces

Namespaces in Kubernetes provide a mechanism for isolating groups of resources within a single cluster. They are particularly useful for organizing resources into logical groups, setting various kinds of security policies, quotas, and more. By default, a Kubernetes cluster comes with the following namespaces:

- **default**: The default namespace for objects with no other namespace.
- **kube-system**: The namespace for objects created by the Kubernetes system.
- **kube-public**: A namespace that is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage.

Why Use Namespaces?

Namespaces are beneficial when you have:

- Multiple environments (development, staging, production) in a single cluster.
- Multiple teams or projects sharing a cluster.
- A need for resource isolation and management.

Namespace Commands

- To view existing namespaces: **kubectl get namespaces**
- To create a new namespace: **kubectl create namespace <namespace-name>**

Namespace in YAML Definition

You can specify a namespace in the metadata of your resource definition file:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: dev
spec:
  containers:
  - name: nginx
    image: nginx
```

Viewing and Managing Namespaces

- To view resources within a specific namespace: **kubectl get all -n <namespace-name>**

- To view all resources in all namespaces: **kubectl get all --all-namespaces**
- To view services in a specific namespace: **kubectl get svc -n <namespace-name>**

Example Workflow

1. **Create Namespace**: Let's create a namespace called **dev**. : **kubectl create namespace kubecart**
2. **Run a Pod in the New Namespace**: **kubectl run nginx --image=nginx --namespace=kubecart**

If you try to create a pod with the same name in the default namespace, it will work since the namespaces are isolated: **kubectl run nginx --image=nginx --namespace=default**

But if u try to create a new pod with same name in same namespace it will reflect error

3. **Create Resources from a YAML File**:

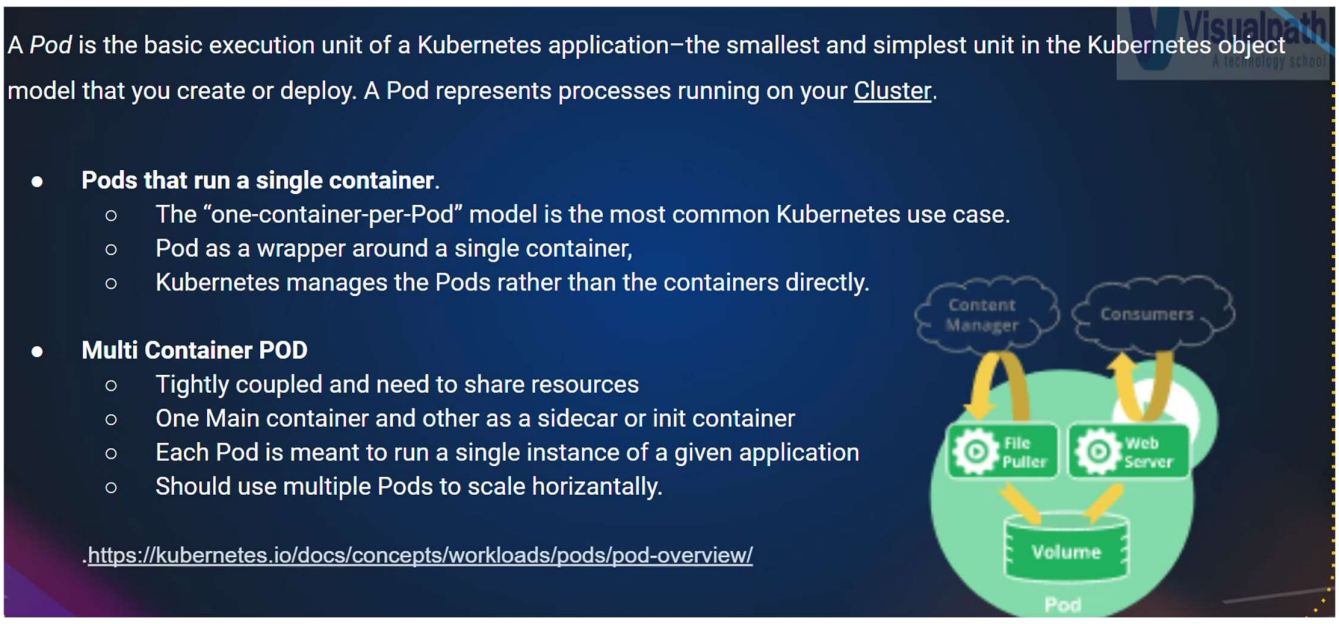
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: dev
spec:
  containers:
  - name: nginx
    image: nginx
```

4. Apply the configuration: **kubectl apply -f pod.yaml**
5. **View Pods in a Namespace**: **kubectl get pods -n kubecart**
6. **Delete a Namespace**:

- Deleting a namespace will remove all resources within it: **kubectl delete namespace kubecart**

Be cautious, as this command deletes everything in the namespace.

Pods

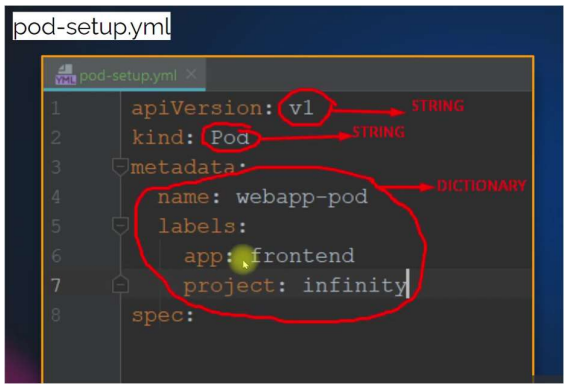


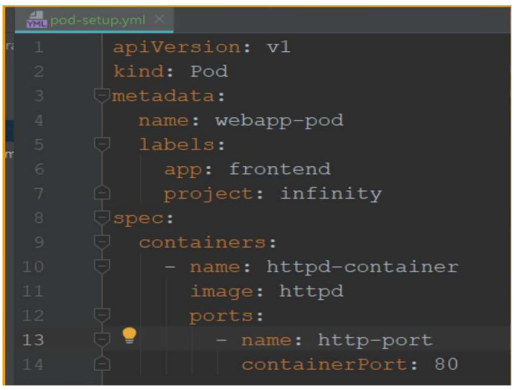
What is a Pod?

A **pod** is the smallest deployable unit in Kubernetes. It represents a single instance of a running process in your cluster. Each pod can contain one or more containers, usually related and working together. However, the most common practice is to run a single container per pod. In cases where multiple containers are used within a single pod, they typically act as helper containers, such as sidecar or init containers, which assist the main container.

Key Concepts

- **Single Container per Pod**: The most common use case.
- **Multi-Container Pod**: Containers share resources (network, storage) and usually include helper containers.
- **Pod Lifecycle**: Kubernetes manages pods, not individual containers. Commands and operations are executed on pods.





Components of a Pod Definition

- **apiVersion**: The version of the Kubernetes API used.
- **kind**: The type of object, in this case, **Pod**.
- **metadata**: Metadata about the pod, including name and labels.
- **spec**: The specification of the pod, including containers and their properties.

Creating and Managing Pods

Creating a Pod

- You can create a pod using the **kubectl create -f** command with a YAML definition file: **kubectl create -f mypod.yaml**

Viewing Pods

- To view the status of pods: **kubectl get pods**
- To view pods across all namespaces: **kubectl get pods --all-namespaces**

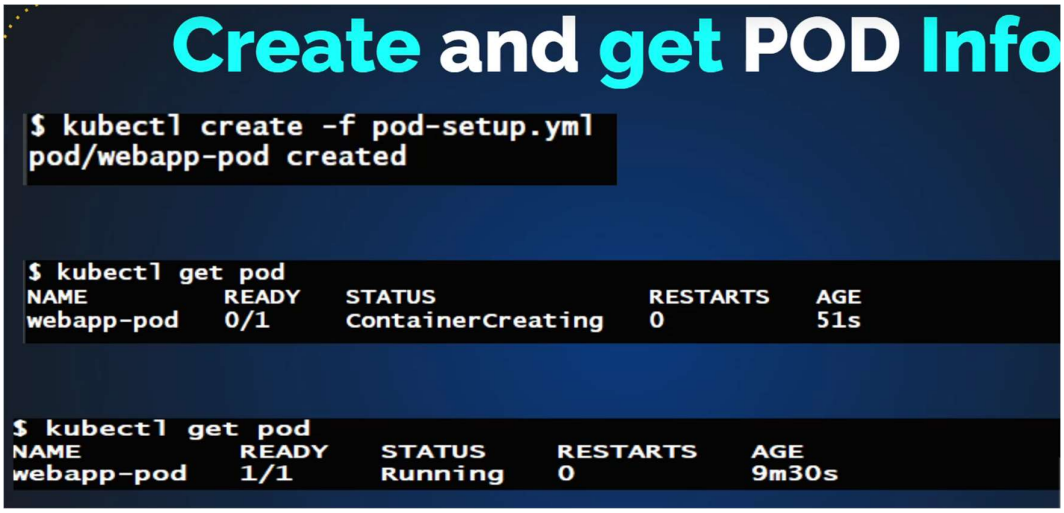
Describing a Pod

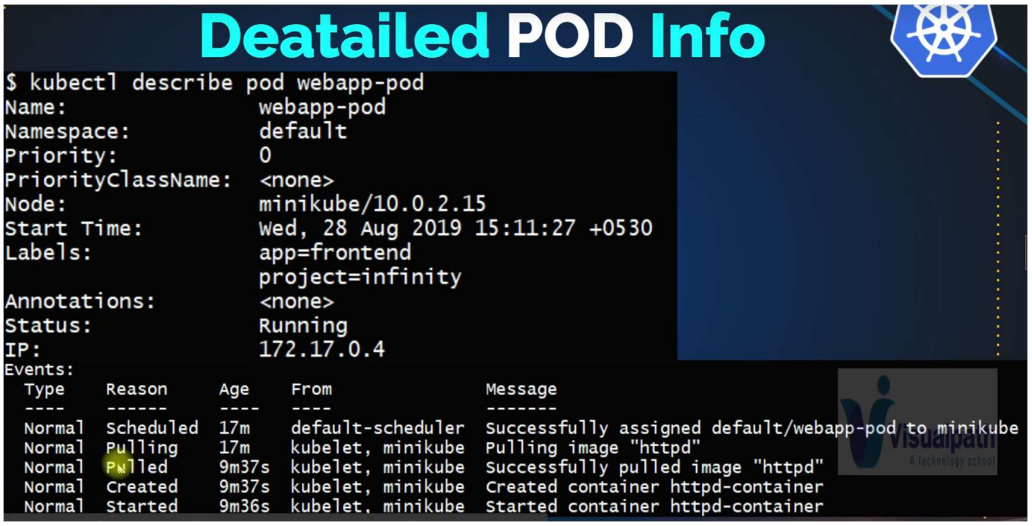
- To get detailed information about a pod: **kubectl describe pod mypod**
- This command provides extensive details, including events that can help in troubleshooting.

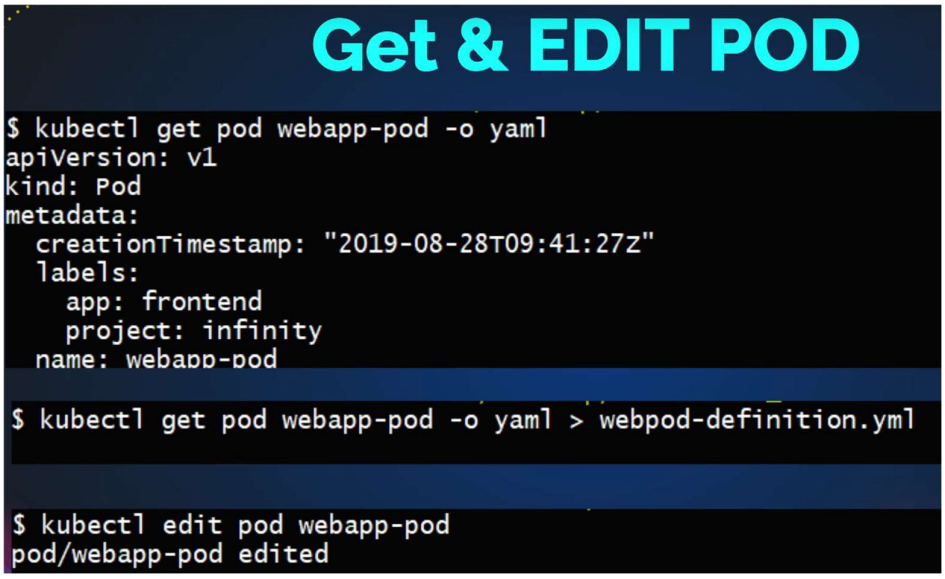
Editing a Pod

- While you can edit some aspects of a running pod, it's limited. Use **kubectl edit** to edit a pod: **kubectl edit pod mypod**

However, not all fields are editable, and it's often better to delete and recreate the pod with updated definitions.







Practical Example:

- mkdir definitions/pods
- cd definitions/pods

- vim vproapppod.yaml
- enter following 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
~
~
```

- kubectl create -f vproapppod.yaml

```
root@ip-172-31-39-99:~/definitions/pod# kubectl get pod
NAME      READY   STATUS              RESTARTS   AGE
vproapp   0/1     ContainerCreating   0          4s
```

- kubectl get pod

```
root@ip-172-31-39-99:~/definitions/pod# kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
vproapp   1/1     Running   0          59s
root@ip-172-31-39-99:~/definitions/pod# kubectl describe pod vproapp
```

-

Different levels of logging

Troubleshooting Kubernetes Pods

Welcome! In this lecture, we will discuss how to find and resolve issues with your Kubernetes pods. Mistakes are inevitable, but learning to identify and fix them is crucial. Here, we'll walk through some common issues and the methods to troubleshoot them.

Setting Up a Local Environment

Before diving into troubleshooting, it's essential to have a local environment for testing. This helps to catch and fix errors early before deploying them in production. Make sure your project allows setting up a local environment, which can be done on VMs, local instances, or any setup you prefer.

Common Pod Issues

Example Scenario

Let's start with a Kubernetes cluster where we have three pods: **kubectl get pods**

Output:

NAME	READY	STATUS	RESTARTS	AGE
nginx1	1/1	Running	0	5m
nginx12	0/1	ImagePullBackOff	0	3m
web2	0/1	CrashLoopBackOff	8	2m

Here, we see two problematic pods: **nginx12** and **web2**.

Identifying Image Issues

ImagePullBackOff

The **nginx12** pod is in an **ImagePullBackOff** state. This usually indicates a problem with the image name or tag. To investigate further:

1. **Check the Image Name**: kubectl get pod nginx12 -o yaml
    - Look for the image field under the container spec:

```
containers:
- name: nginx
  image: ngnox:latest
```

    - Here, **ngnox** is a typo. The correct image name should be **nginx**.

2. **Describe the Pod**: **kubectl describe pod nginx12**
    - In the events section, you should see an error message like:
    - Failed to pull image "ngnox:latest": rpc error: code = Unknown desc = Error response from daemon: pull access denied for ngnox, repository does not exist or may require 'docker login': denied: requested access to the resource is denied
    - Apply the corrected definition:
      kubectl delete pod nginx12
      kubectl apply -f pod.yaml

Resolving CrashLoopBackOff

The **web2** pod is in a **CrashLoopBackOff** state. This indicates the pod is crashing and Kubernetes is attempting to restart it. To investigate:

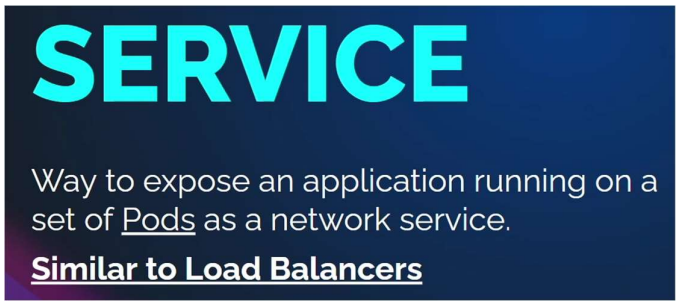
1. **Check Pod Logs**:

- The most effective way to understand why a pod is crashing is to check its logs: **kubectl logs web2**

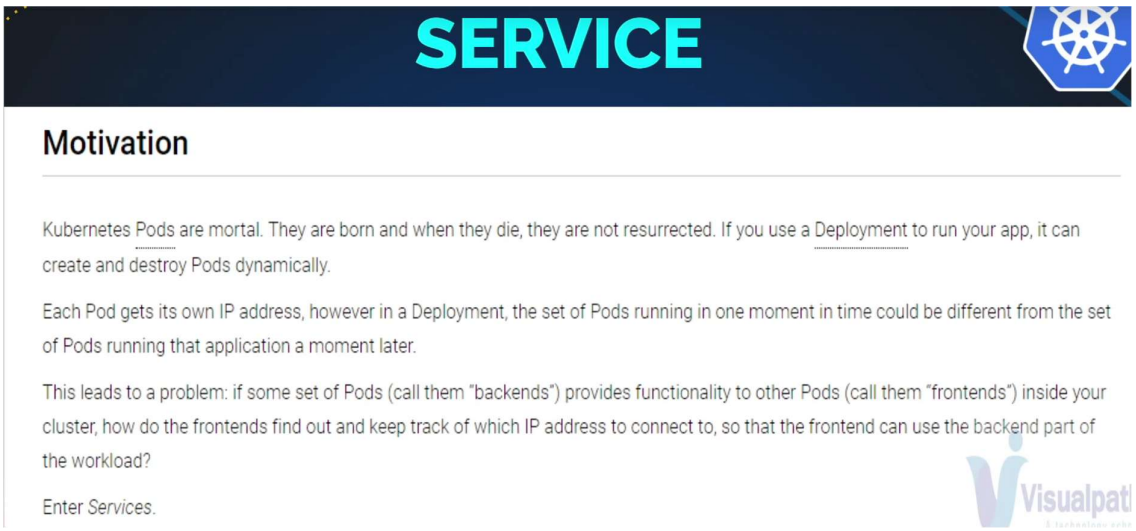
```
ubuntu@ip-172-31-12-44:~$ kubectl logs web2
/docker-entrypoint.sh: 38: exec: test47: not found
ubuntu@ip-172-31-12-44:~$ history | grep test47
   86  kubectl run web2 --image=nginx test47
  120  history | grep test47
ubuntu@ip-172-31-12-44:~$ kubectl delete pod web2
pod "web2" deleted
ubuntu@ip-172-31-12-44:~$ kubectl run web2 --image=nginx
pod/web2 created
ubuntu@ip-172-31-12-44:~$ kubectl get pod
NAME      READY   STATUS    RESTARTS      AGE
nginx1    1/1     Running   1 (95m ago)   97m
nginx12   1/1     Running   0             5m4s
web2      1/1     Running   0             5s
ubuntu@ip-172-31-12-44:~$
```

- kubectl delete pod web2
  kubectl run web2 --image=nginx

Service



Pods do not have elastic IP so we use services for different purposes.



In k8s there are 3 types of services :

1. NodePort
2. ClusterIP
3. LoadBalancer

NodePort: It is similar to port mapping. We have seen in docker a host port you pick up and you map it with the container port. It's exactly the same way. And mostly it's for nonprod purpose, not for production, not for exposing the frontend to the production. And only this is to expose your pod to the outside network. That's why you can use node port.

Use Cases

- Suitable for non-production environments.
- Exposing a service for testing purposes.

ClusterIP: if you don't want to expose it to outside network but internal like Tomcat connecting to MySQL. So for MySQL you need a static endpoint so you can create a service of type cluster IP. Cluster IP for internal communication. So here there will be no port mapped to your node internal communication.

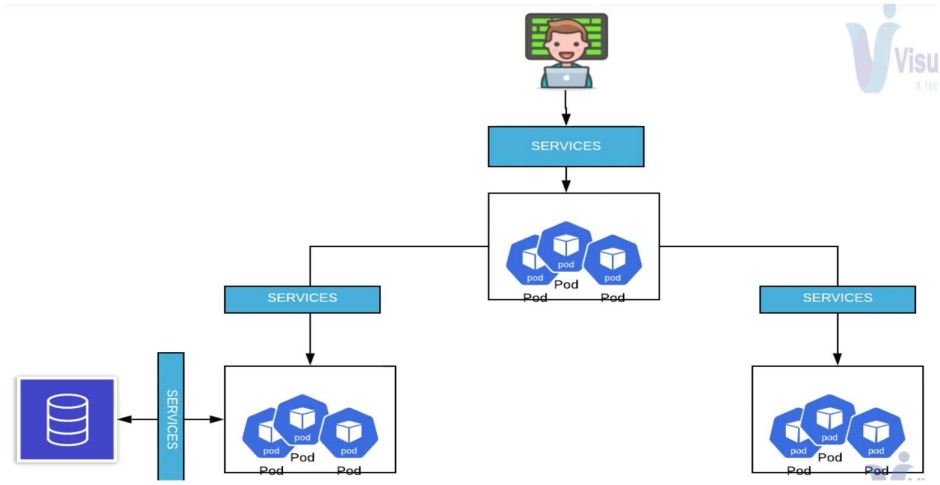
Use Cases

- Internal communication within the cluster.
- Services that do not need to be exposed to the outside world.

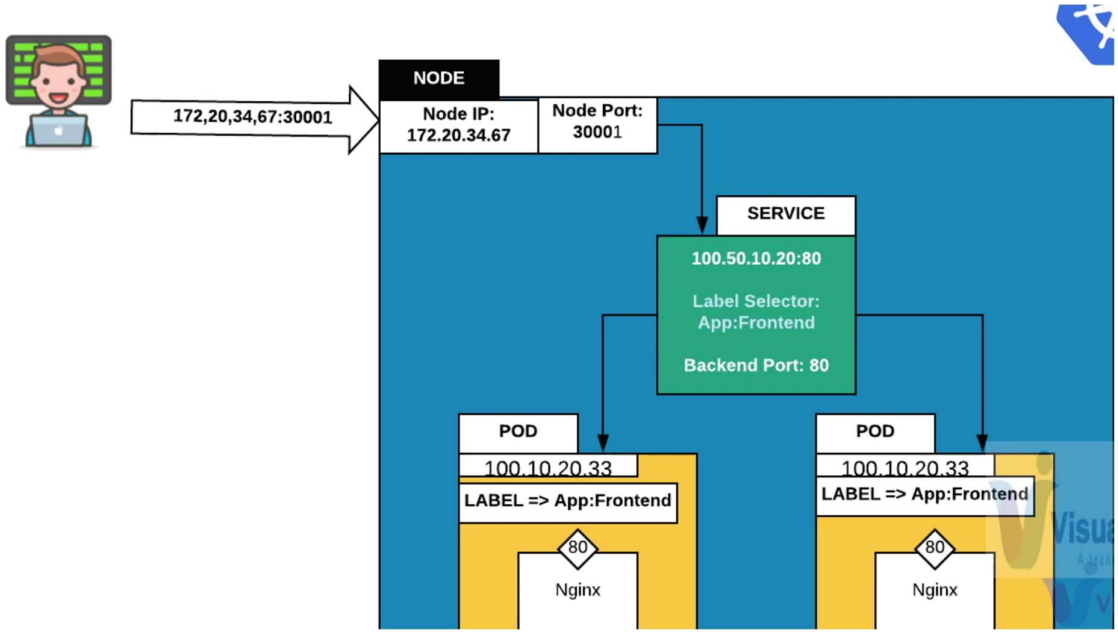
LoadBalancer: It is again to expose to the outside network for production use cases. We are running a Tomcat pod and I want users from the internet to access that. I will create the service of type load balancer and on AWS, it's going to create actually an elastic load balancer and map my Pod to it.

Use Cases

- Exposing services to the internet.
- Suitable for production environments.



So every pod or a cluster of pod that are running, you need service in front of that if it's a network, if it's providing a network service, okay, like a frontend, we will have a service for back end RabbitMQ Elastic Cache you'll have again service in front of it. MySQL, you'll have a service in front of it. That's how the communication happens between pods and between user to the pods.



First, we'll understand service of type node port.

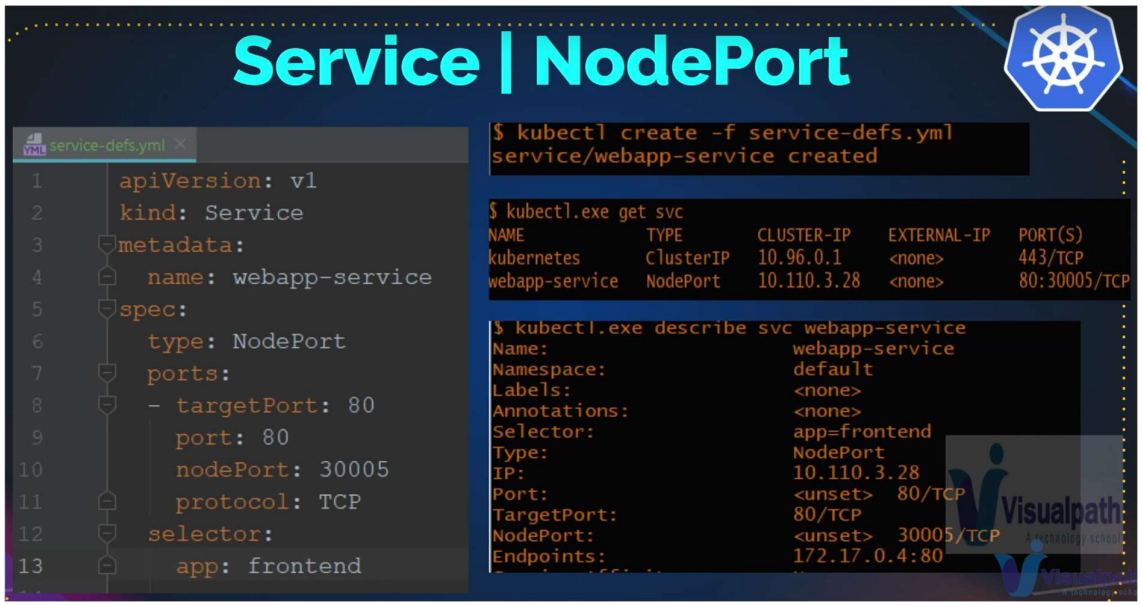
Node port.

you have two pods running in a node, that blue colour rectangle is your node worker node and you have two pods running over here. Pod will have label and Pod will have IP address and of course, the container running inside that. So, the nginx container is running on both the pods, one container per Pod and it's exposed on port 80. Okay, pod has an IP address and a port number to access and it has a label. Now, service will be like a

load balancer. Service will have a static IP address, which will not change until unless you don't delete it. It will have a frontend port(100.50.10.20:80) like a load balancer has a frontend port. It's for internal communication. And it has backend port(80) and the back end port will be the port number of the container, obviously. Okay, so Service, like a load balancer, has a frontend port and a back end port.

But how does it know which Pod it has to route the request to?

There could be hundreds of pods running like this. Well, it is going to match the label selector. When we create a service, we give label selector. We give a label name. Any Pod that has this label **app frontend**, it's going to forward the request to that Pod on port 80. So I have two part. If I run exactly third part, exactly same with same label, my third part will be automatically included under the Service auto discovery. Okay, this is internal. When you say node port, you have to mention a node port number. So let's say 30,001(shown in above fig). So a node port or host port 30,001 will be mapped to your service. When you access the node by giving its IP address and the node port, the request will be forwarded to the Service. The Service is going to forward the request to the Pod and Pod will send it to the container. And the same way it comes back observe. There are multiple port over here. 30,001 is the node port for the outside communication outside network. It sends it to the Service. Service has an internal frontend port 80. It's going to send the request on the back end port 80, which is the container port. Okay, so your node port and your service internal front end port can be same or can be different. Back end port and the container port should be exactly same. And label selector should match with the label of the Pod. So when you're creating a service, two things are very important. Matching the label selector and the back end port. That is the service of type node port




```
root@ip-172-31-39-99:~/definitions/app# cat 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
```

here the target port is number based we can also make it text based

vim vproapp-nodeport.yaml

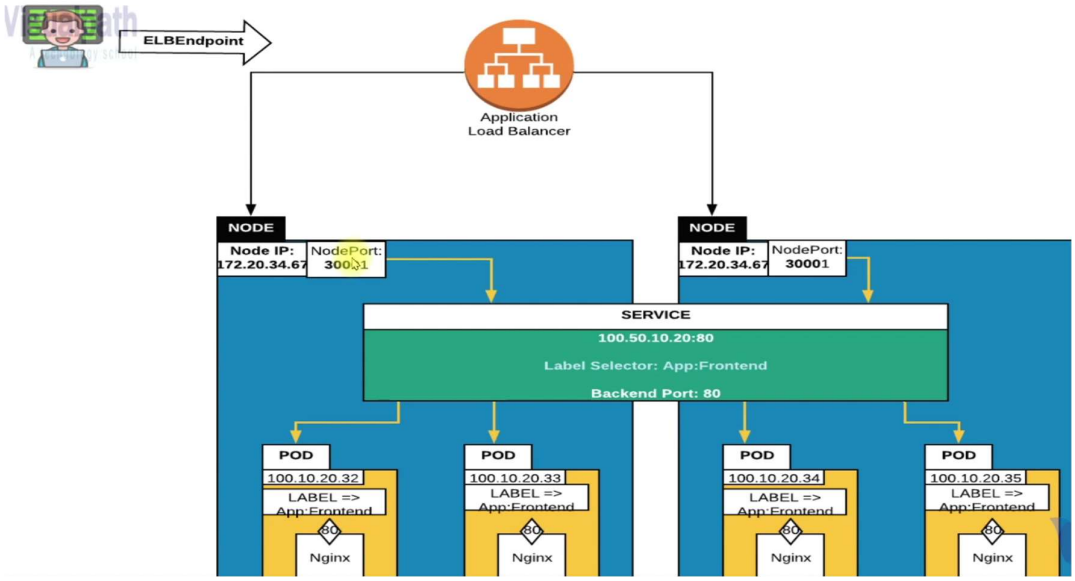
```
apiVersion: v1
kind: Service
metadata:
  name: helloworld-service
spec:
  ports:
  - port: 8090
    nodePort: 30001
    targetPort: vproapp-port
    protocol: TCP
  selector:
      app: vproapp
  type: NodePort
~
```

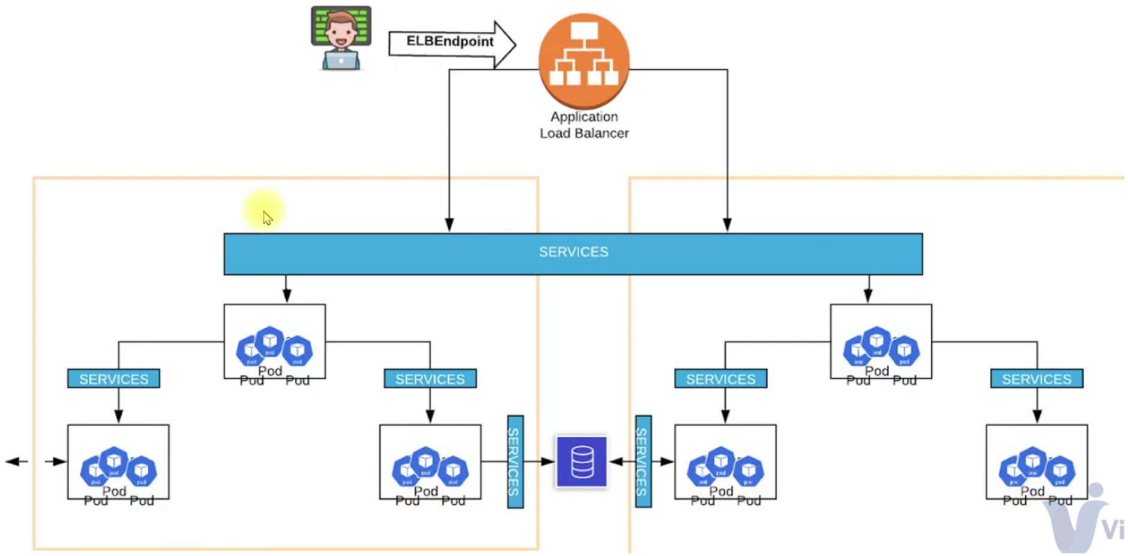
now enter **kubectl create -f vproapp-nodeport.yaml**

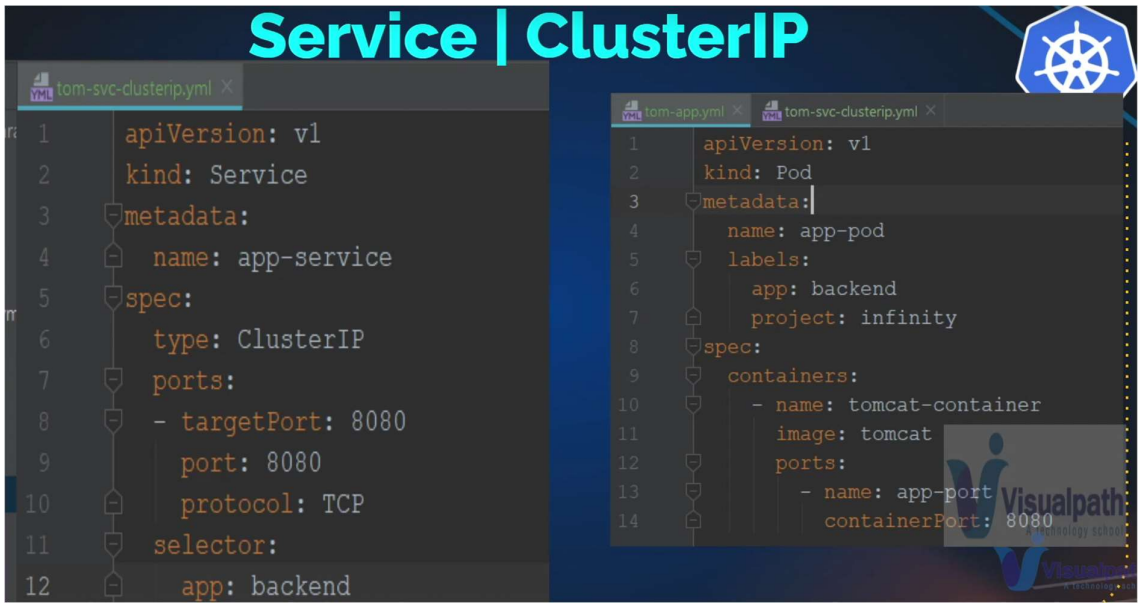
```
root@ip-172-31-39-99:~/definitions/app# kubectl get svc
NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
helloworld-service   NodePort    100.66.245.247   <none>        8090:30001/TCP   6s
kubernetes           ClusterIP   100.64.0.1       <none>        443/TCP          51m
root@ip-172-31-39-99:~/definitions/app#
```

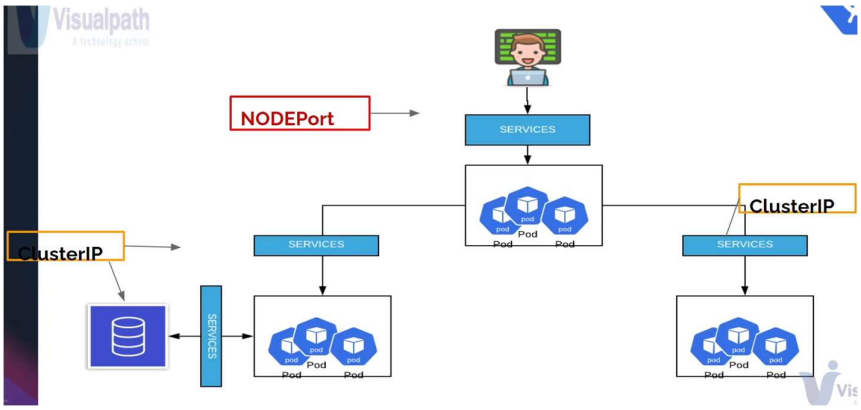
Load Balancer Service

For this we just change the type from nodeport to loadbalncer and port to 80 from 8090









Replicaset

Kubernetes ReplicaSet

ReplicaSet is a Kubernetes object that ensures a specified number of pod replicas are running at any given time. It is often used to guarantee the availability of a certain number of identical pods.

Purpose:

1. **High Availability:** Ensures your application is always available by maintaining the desired number of replicas.
2. **Auto-healing:** Automatically recreates pods if they fail or are deleted.
3. **Scalability:** Allows you to easily scale the number of pod replicas up or down.

How it Works:

- A ReplicaSet watches over a set of pods and ensures that the desired number of replicas are running.
- If a pod fails, the ReplicaSet creates a new one to replace it.
- If a node goes down, the ReplicaSet ensures that the pods running on that node are recreated on other available nodes.

Example of replicaset.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
```

```
  spec:
    containers:
    - name: php-redis
      image: us-docker.pkg.dev/google-samples/containers/gke/gb-frontend:v5
```

Explanation:

- **apiVersion:** Specifies the API version.
- **kind:** Specifies the type of the object, which is ReplicaSet in this case.
- **metadata:** Contains the name of the ReplicaSet.
- **spec:** Contains the specification of the ReplicaSet.
    - **replicas:** The number of desired replicas.
    - **selector:** Specifies how to identify the pods that belong to this ReplicaSet. The **matchLabels** section should match the labels defined in the pod template.
    - **template:** Describes the pod that will be created by this ReplicaSet.
        - **metadata:** Contains labels that must match the selector.
        - **spec:** Describes the containers within the pod.

Managing ReplicaSets

Creating a ReplicaSet:

- kubectl apply -f replicaset.yaml

Checking ReplicaSets and Pods:

- kubectl get rs
- kubectl get pods

Deleting a ReplicaSet:

- kubectl delete rs frontend

Scaling ReplicaSets

Scaling Using YAML:

Update the **replicas** field in the ReplicaSet YAML definition and reapply it:

```
spec:
  replicas: 5
```

Then apply the changes:

- kubectl apply -f replicaset.yaml

Scaling Using Command Line:

- kubectl scale --replicas=5 rs/frontend

Editing a ReplicaSet Directly:

You can also edit the ReplicaSet directly:

- kubectl edit rs frontend

Find the **replicas** field and update it. Save and exit the editor, and Kubernetes will update the number of replicas accordingly.

Practical Example

Replicaset.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

Kubectl create -f replicaset.yaml

Kubectl get rs

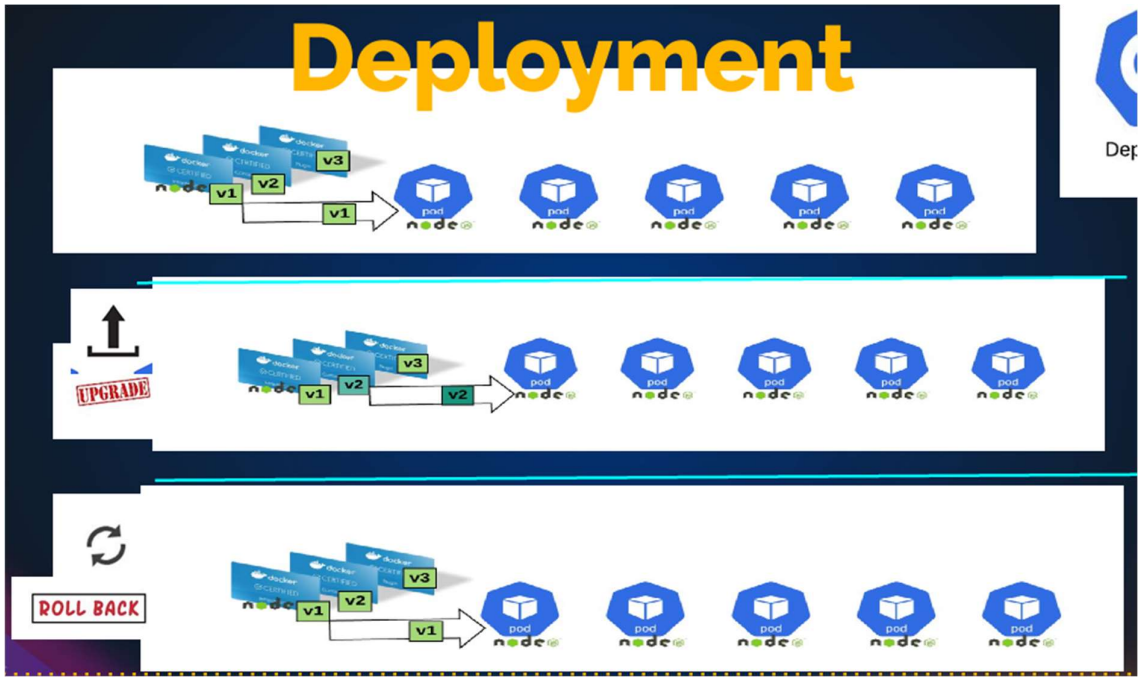
```
ubuntu@ip-172-31-12-44: $ kubectl get rs
NAME       DESIRED   CURRENT   READY   AGE
frontend   3         3         3       41s
ubuntu@ip-172-31-12-44:~$ kubectl get pod
NAME             READY   STATUS    RESTARTS   AGE
frontend-lqvrq   1/1     Running   0          51s
frontend-qmxml   1/1     Running   0          51s
frontend-s4kbp   1/1     Running   0          51s
ubuntu@ip-172-31-12-44:~$ kubectl delete pod frontend-qmxml frontend-s4kbp
pod "frontend-qmxml" deleted
pod "frontend-s4kbp" deleted
ubuntu@ip-172-31-12-44:~$ kubectl get pod
ME             READY   STATUS    RESTARTS   AGE
rontend-lqvrq   1/1     Running   0          2m41s
frontend-nh7kf   1/1     Running   0          4s
frontend-tbvdp   1/1     Running   0          4s
ubuntu@ip-172-31-12-44:~$
```

```
ubuntu@ip-172-31-12-44:~$ kubectl get pod
NAME             READY   STATUS    RESTARTS   AGE
frontend-lqvrq   1/1     Running   0          3m38s
frontend-nh7kf   1/1     Running   0          61s
frontend-tbvdp   1/1     Running   0          61s
frontend-xccrw   1/1     Running   0          7s
frontend-xtdx7   1/1     Running   0          7s
ubuntu@ip-172-31-12-44:~$ kubectl scale --replicas=1 rs/frontend
replicaset.apps/frontend scaled
ubuntu@ip-172-31-12-44:~$ kubectl get pod
NAME             READY   STATUS    RESTARTS   AGE
frontend-lqvrq   1/1     Running   0          6m22s
ubuntu@ip-172-31-12-44:~$ kubectl edit rs frontend
replicaset.apps/frontend edited
ubuntu@ip-172-31-12-44:~$ kubectl get pod
NAME             READY   STATUS    RESTARTS   AGE
frontend-gdqr6   1/1     Running   0          2s
frontend-lqvrq   1/1     Running   0          7m1s
ubuntu@ip-172-31-12-44:~$ kubectl delete rs frontend
replicaset.apps "frontend" deleted
```

https://kubernetes.io/docs/reference/kubectl/quick-reference/

Deployment

- A Deployment controller provides declarative updates for Pods and
- ReplicaSets.
- Define desired state in a Deployment, and the Deployment controller
- changes the actual state to the desired state at a controlled rate.
- Deployment creates ReplicaSet to manage number of PODS.



Kubernetes Deployment

In this lecture, we'll delve into Kubernetes Deployments, one of the most commonly used objects by DevOps teams. Deployments provide a way to manage and automate the deployment of applications in a Kubernetes cluster, including updating, scaling, and rolling back applications.

Key Concepts:

1. **Declarative Updates:** With Deployments, you can declare the desired state of your application (e.g., the number of replicas, the container image version) in a YAML file. Kubernetes will then manage the transition from the current state to the desired state.
2. **Rolling Updates:** When you update your application, the Deployment controller updates the pods in a rolling fashion, ensuring that the application remains available during the update process.
3. **Rollback:** If something goes wrong during an update, you can easily roll back to a previous version.

Sample Deployment.yaml looks like this

```
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
```

Explanation:

- **apiVersion:** Specifies the API version (apps/v1 for Deployments).
- **kind:** Specifies the type of object (Deployment).
- **metadata:** Contains the name and labels for the Deployment.
- **spec:** Defines the desired state of the Deployment.
    - **replicas:** The desired number of pod replicas.
    - **selector:** Defines how to identify the pods managed by this Deployment.
    - **template:** Describes the pod to be created.
        - **metadata:** Labels that must match the selector.
        - **spec:** Specifies the containers within the pod, including the image and ports.

Managing Deployments

Create a Deployment:

- kubectl apply -f deployment.yaml

Check Deployment Status:

- kubectl get deployments kubectl get pods kubectl get rs

Update a Deployment:

To update the container image to a new version:

- kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1

This command updates the Deployment to use the new image, which triggers a rolling update.

Rollback a Deployment:

To rollback to the previous version:

- kubectl rollout undo deployment/nginx-deployment

You can also rollback to a specific revision:

- kubectl rollout undo deployment/nginx-deployment --to-revision=2

View Rollout History:

- kubectl rollout history deployment/nginx-deployment

Scaling Deployments

To scale the number of replicas:

- kubectl scale deployment/nginx-deployment --replicas=5

Kubernetes Deployments are essential for managing applications in a cluster. They provide powerful features like declarative updates, rolling updates, and rollbacks, making it easier to maintain and scale applications. Always prefer to manage your deployments through YAML definition files, especially in production environments, to ensure consistency and version control. Practice creating, updating, and rolling back Deployments to become comfortable with these operations.

Commands & Arguments

Kubernetes: Commands and Arguments in Pods

In this lecture, we will explore how to pass commands and arguments to containers within Kubernetes pods. Remember, it is the container inside the pod that executes the commands, not the pod itself.

Container Commands and Arguments in Docker

When building Docker images, you can specify what command the container will run using the **CMD** or **ENTRYPOINT** directives in the Dockerfile.

Dockerfile Examples:

1. **Using CMD:**

- FROM ubuntu CMD ["echo", "Hello, World!"]

When you run this container, it will execute **echo "Hello, World!"**.

2. **Using ENTRYPOINT:**

- FROM ubuntu ENTRYPOINT ["echo"]

When you run this container, you need to provide the argument:

- docker run my-image "Hello, World!"

This will execute **echo "Hello, World!"**.

3. **Using Both ENTRYPOINT and CMD:**

- FROM ubuntu ENTRYPOINT ["echo"] CMD ["Hello, World!"]

Running the container without arguments:

- docker run my-image

This will execute **echo "Hello, World!"**.

Overriding the argument:

- docker run my-image "Hello, Kubernetes!"

This will execute **echo "Hello, Kubernetes!"**.

Commands and Arguments in Kubernetes Pods

In Kubernetes, you can specify commands and arguments for containers in the pod definition file.

Pod Definition Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: command-demo
spec:
  containers:
  - name: my-container
    image: debian
    command: ["printenv"]
    args: ["HOSTNAME", "KUBERNETES_PORT"]
```

Explanation:

- **command:** Specifies the command to run in the container.
- **args:** Specifies the arguments to pass to the command.

Creating and Applying the Pod Definition

1. Create a YAML file**:**
2. paste the pod yaml file from above
3. apply the yaml file
    - kubectl apply -f com.yaml
4. check pod status
    - kubectl get pods
    - You may see the status as **Completed**, indicating that the command has finished executing.
5. **View the logs to see the command output:**
    - kubectl logs command-demo
    - The output will show the values of the environment variables **HOSTNAME** and **KUBERNETES_PORT**.

Using Environment Variables

You can define environment variables and use them in commands.

Pod Definition with Environment Variables:

```
env:
  - name: MESSAGE
    value: "Hello, World!"
  command: ["echo"]
  args: ["$(MESSAGE)"]
```

Explanation:

1. **env:** Defines environment variables for the container.
2. **command:** Specifies the command to run (**echo**).
3. **args:** Passes the environment variable as an argument to the command.

The output will show **Hello, World!**.

Volumes

Kubernetes Volumes

In this lecture, we'll discuss how to use volumes in Kubernetes. Volumes in Kubernetes provide a way for containers to persist data and share it between multiple containers. There are many types of volumes supported by Kubernetes, and while persistent volumes and their backend storage solutions are a more advanced topic, it's essential for DevOps engineers and developers to understand how to map a volume to a pod.

Types of Volumes Supported by Kubernetes

Kubernetes supports several types of volumes, including:

- AWS Elastic Block Store (EBS)
- Azure Disk
- Ceph
- Cinder
- Fibre Channel (FC)
- Flocker
- Google Cloud Persistent Disk (GCE Persistent Disk)
- GlusterFS
- HostPath
- NFS
- Portworx
- VMware vSphere Volume

Each of these storage solutions may be used depending on your project's requirements and the infrastructure your organization uses.

HostPath Volume Example

In this example, we'll use a HostPath volume, which maps a directory on the worker node to the container's filesystem.

Pod Definition File:

```
apiVersion: v1
kind: Pod
metadata:
  name: dbpod
spec:
  containers:
  - name: mysql
    image: mysql:5.7
    volumeMounts:
    - mountPath: /var/lib/mysql
      name: dbvol
  volumes:
  - name: dbvol
    hostPath:
      path: /data
      type: DirectoryOrCreate
```

Explanation:

- **volumeMounts:** Specifies where the volume will be mounted inside the container.

- **volumes:** Defines the volume and its source. In this case, it's a HostPath volume, meaning it uses a directory on the worker node.
- **hostPath:** Specifies the directory on the worker node (**/data**) and indicates that if the directory doesn't exist, it should be created (**DirectoryOrCreate**).

Applying the Pod Definition

1. **Create a YAML file:**
2. Enter the content as above
3. Apply by kubectl apply -f com.yaml
4. Check the pods by kubectl get pods
5. Describe the pod and u will see like following

```
Volumes:
  dbvol:
    Type:          HostPath (bare host directory volume)
    Path:          /data
    HostPathType:  DirectoryOrCreate
...
Mounts:
  /var/lib/mysql from dbvol (rw)
...
```

Using volumes in Kubernetes is a critical skill for managing stateful applications. HostPath volumes are useful for local testing and development but are not suitable for production due to their lack of redundancy and scalability. For production environments, consider using more robust solutions like AWS EBS, Google Persistent Disk, or network filesystems like NFS or GlusterFS.

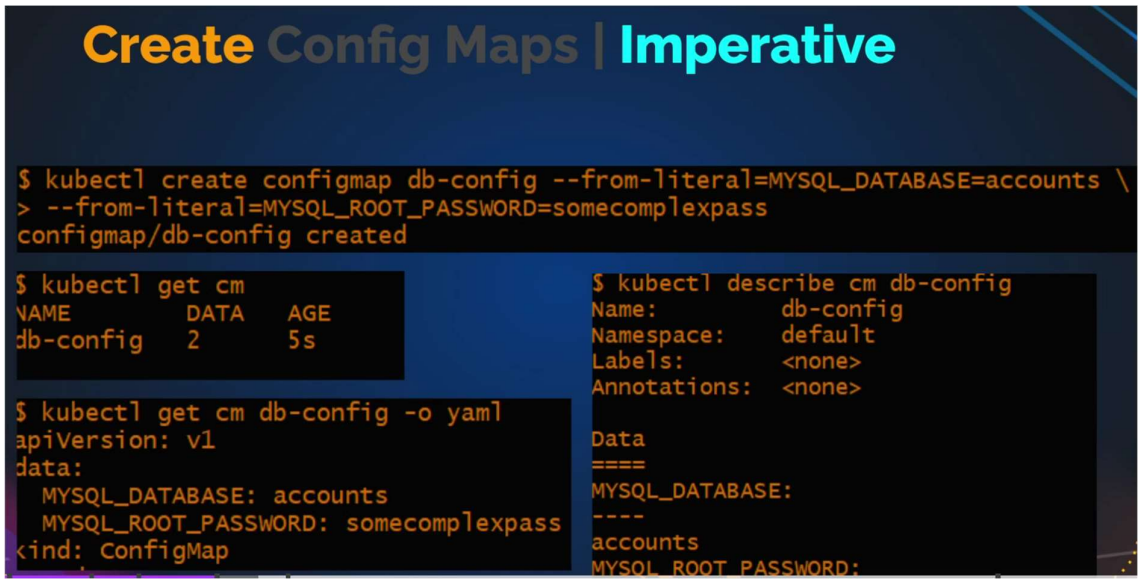
In production scenarios, persistent volumes (PVs) and persistent volume claims (PVCs) provide a more flexible and powerful way to manage storage. As a DevOps engineer or developer, understanding how to map these volumes to your pods ensures that your applications can manage data correctly and persist it across pod restarts.

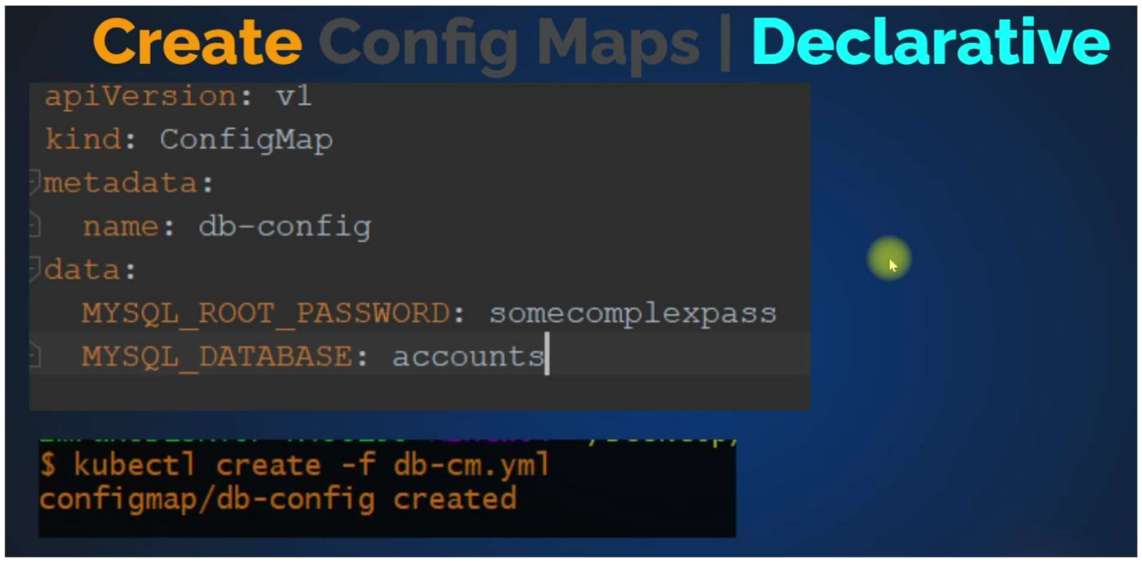
For hands-on practice, try different types of volumes based on your infrastructure and understand how to configure and manage them. This foundational knowledge will be crucial as you work with stateful applications in Kubernetes.

Config Map

Refer to => https://kubernetes.io/docs/concepts/configuration/configmap/

2 types of config map – imperative and declarative





Now Pod reading our config map

Our configmap.yaml is as follows:

```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
```

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

1. Inside a container command and args
2. Environment variables for a container
3. Add a file in read-only volume, for the application to read
4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

Here's an example Pod that uses values from game-demo to configure a Pod:

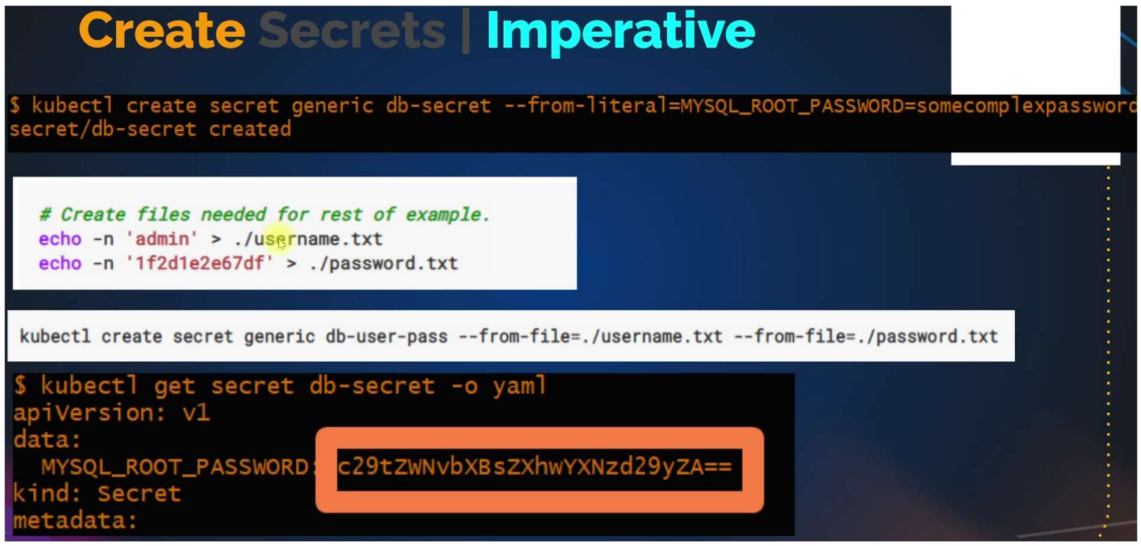
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
                                     # 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"
```

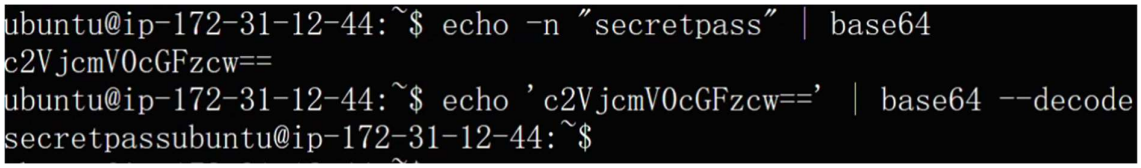
read it propeely to understand and run the following commands

```
ubuntu@ip-172-31-12-44:~$ kubectl get cm
NAME               DATA   AGE
game-demo          4      16m
kube-root-ca.crt   1      75m
ubuntu@ip-172-31-12-44:~$ vim readcmpod.yaml
ubuntu@ip-172-31-12-44:~$ kubectl apply -f readcmpod.yaml
pod/configmap-demo-pod created
ubuntu@ip-172-31-12-44:~$ kubectl get pod
NAME                READY   STATUS    RESTARTS   AGE
configmap-demo-pod   1/1     Running   0          23s
ubuntu@ip-172-31-12-44:~$ kubectl exec --stdin --tty configmap-demo-pod -- /bin/sh
/ # ls /config/
game.properties            user-interface.properties
/ # cd /config/
/config # cat game.properties
enemy.types=aliens,monsters
player.maximum-lives=5
/config # cat user-interface.properties
color.good=purple
color.bad=yellow
allow.textmode=true
/config # echo $PLAYER_INITIAL_LIVES
3
/config # echo $UI_PROPERTIES_FILE_NAME
user-interface.properties
```

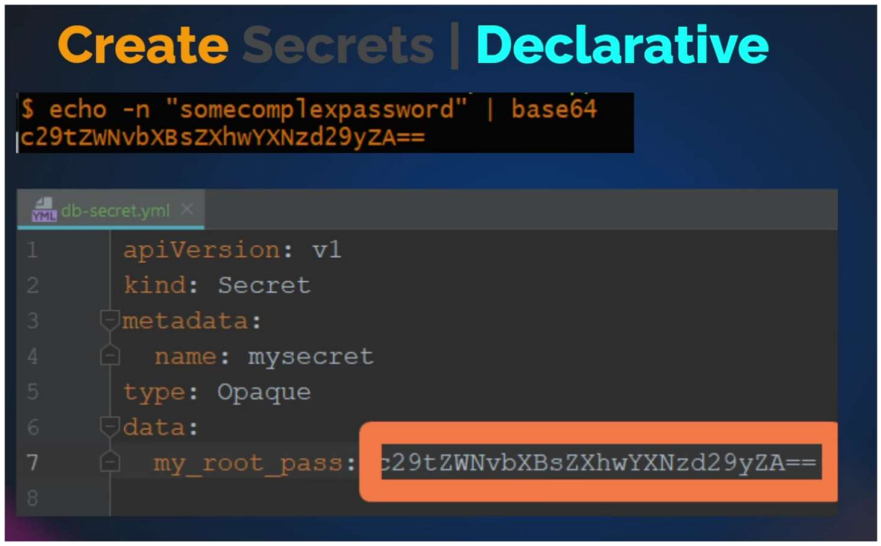
Secrets

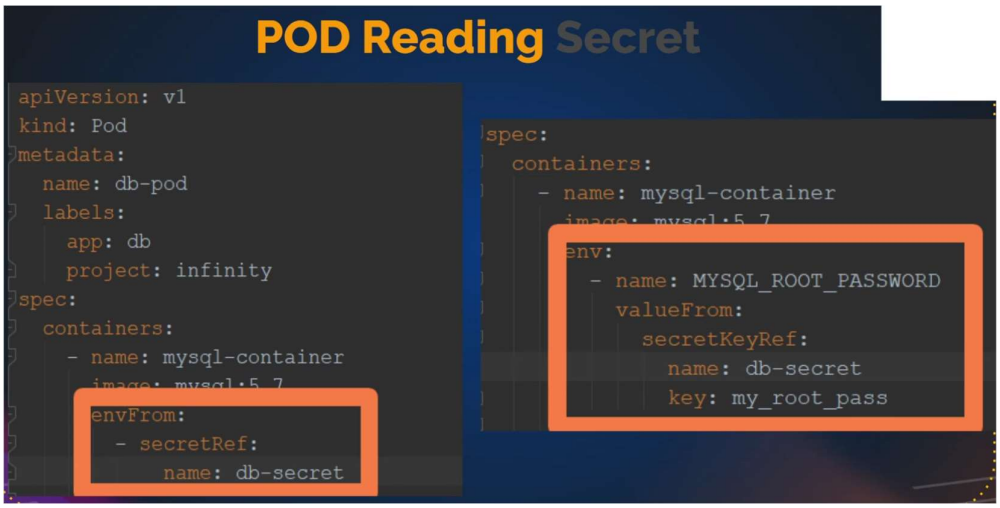
- Shares encoded/encrypted variables to our POD
- Stores and manages sensitive info, such as passwords





U can decode ur data using base64





For secret firstly we need to encrypt our data then add it to our secret file then connect it to our pod

There are many types of secret

```
ubuntu@ip-172-31-12-44: $ echo -n "admin" | base64
YWRtaW4=
ubuntu@ip-172-31-12-44:~$ echo -n "mysecretpass" | base64
bXlzZWNyZXRwYXNz
ubuntu@ip-172-31-12-44:~$ vim mysecret.yaml
ubuntu@ip-172-31-12-44:~$ vim mysecret.yaml
ubuntu@ip-172-31-12-44:~$ vim mysecret.yaml
ubuntu@ip-172-31-12-44:~$ kubectl create -f mysecret.yaml
secret/mysecret created
ubuntu@ip-172-31-12-44:~$ vim readsecret.yaml
ubuntu@ip-172-31-12-44:~$ kubectl create -f readsecret.yaml
pod/secret-env-pod created
ubuntu@ip-172-31-12-44:~$ kubectl get pod
NAME                 READY   STATUS    RESTARTS   AGE
configmap-demo-pod   1/1     Running   0          38m
secret-env-pod       1/1     Running   0          6s
ubuntu@ip-172-31-12-44:~$ kubectl exec --stdin --tty secret-env-pod -- /bin/bash
root@secret-env-pod:/data# echo $username

root@secret-env-pod:/data# echo $SECRET_USERNAME
admin
root@secret-env-pod:/data# echo $SECRET_PASSWORD
mysecretpass
root@secret-env-pod:/data#
```

The mysecret.yaml file is as follows

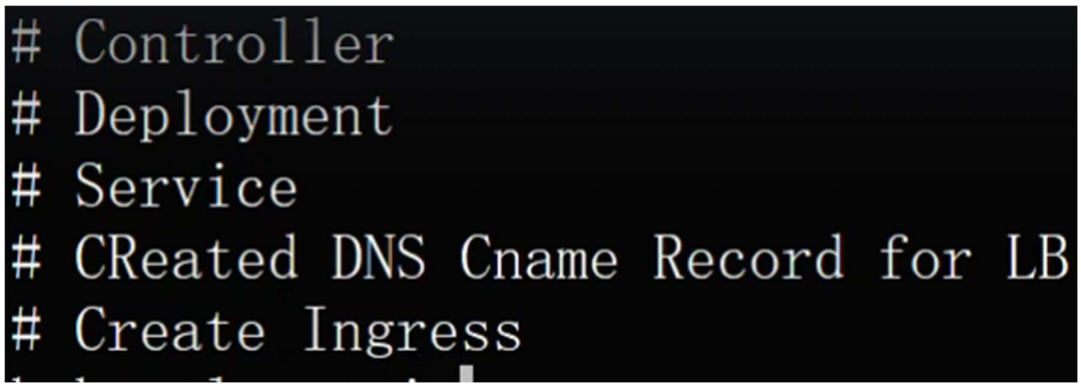
```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  username: YWRtaW4=
  password: bXlzZWNyZXRwYXNz
type: Opaque
~
```

And readsecret.yaml like this

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
            optional: false # same as default; "mysecret" must exist
                            # and include a key named "username"
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
            optional: false # same as default; "mysecret" must exist
                            # and include a key named "password"
  restartPolicy: Never
~
```

Ingress

Workflow as follows



Introduction to Ingress in Kubernetes

In this lecture, we'll cover the concept of ingress in Kubernetes. Ingress is an API object that manages external access to services within a Kubernetes cluster, typically over HTTP or HTTPS.

What is Ingress?

- **Ingress**: It allows external access to HTTP and HTTPS routes to services within a Kubernetes cluster. It can provide functionalities such as load balancing, SSL termination, and name-based virtual hosting.
- **Ingress Controller**: To use ingress, you need an ingress controller, which is a type of load balancer within the Kubernetes environment.

Benefits of Using Ingress

- Manages multiple services with a single entry point.
- Handles routing rules for HTTP/HTTPS traffic.
- Can perform SSL termination, reducing the need for SSL configuration within individual pods.
- Supports various types of routing (e.g., path-based, host-based).

Setting Up Ingress with NGINX

We'll use the NGINX ingress controller, one of the most commonly used controllers.

Steps to Install NGINX Ingress Controller

1. **Install the Ingress Controller**:
    - kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml
2. **Verify Installation**:
    - kubectl get all -n ingress-nginx

```
ubuntu@ip-172-31-12-44:~$ kubectl get all -n ingress-nginx
NAME                                            READY   STATUS      RESTARTS   AGE
pod/ingress-nginx-admission-create-5hl68        0/1     Completed   0          23s
pod/ingress-nginx-admission-patch-lt7lm         0/1     Completed   1          23s
pod/ingress-nginx-controller-69fbbf9f9c-h67pr   1/1     Running     0          23s

NAME                                         TYPE           CLUSTER-IP       EXTERNAL-IP
                                              PORT(S)                      AGE
service/ingress-nginx-controller             LoadBalancer   100.68.173.127   a07bff36074ee45d6a8ab27eb91b1f83-f
c5a0f4a1f500024.elb.us-west-2.amazonaws.com   80:32718/TCP,443:32372/TCP   23s
service/ingress-nginx-controller-admission   ClusterIP      100.66.154.8     <none>
                                              443/TCP                      23s

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ingress-nginx-controller   1/1     1            1           23s

NAME                                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/ingress-nginx-controller-69fbbf9f9c   1         1         1       23s

NAME                                       COMPLETIONS   DURATION   AGE
job.batch/ingress-nginx-admission-create   1/1           3s         23s
job.batch/ingress-nginx-admission-patch    1/1           4s         23s
ubuntu@ip-172-31-12-44:~$
```

- doing ls will return 3 yaml files services, deployment, ingress
- whuch looks like as follows

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp-container
        image: your-image
        ports:
        - containerPort: 8080
```

Service :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: ClusterIP
```

ingress :

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: myapp-service
            port:
              number: 80
```

kubectl apply -f vprodep.yaml , kubectl apply -f vprosvc.yaml

now go to domain provider and add the dns from load balancer

kubectl apply -f vproingress.yaml

now u can access our project from ur domain

Extra

In this lecture, we will explore a few advanced Kubernetes features that might not be commonly used in every project but are nonetheless very important to understand. These features include Taints and Tolerations, Resource Limits and Requests, Jobs, CronJobs, and DaemonSets.

Taints and Tolerations

Taints and Tolerations are used to control which Pods can be scheduled on which nodes. You can think of tainting a node as "painting" it with a specific property. Only Pods that "tolerate" that property can be scheduled on that node.

Example:

To taint a node, use the following command:

- kubectl taint nodes node1 key=value:NoSchedule

This means no Pods can be scheduled on node1 unless they have a toleration for this taint.

To add a toleration to a Pod, include it in the Pod's definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
  tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
```

Resource Limits and Requests

Resource limits and requests help you manage the resources (CPU and memory) used by your Pods.

- **Requests:** Reserve a specific amount of CPU and memory.
- **Limits:** Restrict the maximum amount of CPU and memory that a Pod can use.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-limited-pod
spec:
  containers:
  - name: my-container
    image: my-image
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Jobs and CronJobs

Jobs: Jobs are used to run a container that performs a specific task and then completes.

Example:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: simple-job
spec:
  template:
    spec:
      containers:
      - name: job-container
        image: my-image
        command: ["my-command"]
      restartPolicy: OnFailure
```

CronJobs: CronJobs allow you to run jobs on a schedule, similar to cron jobs in Linux.

Example:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: simple-cronjob
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: cronjob-container
            image: my-image
            command: ["my-command"]
```

```
        restartPolicy: OnFailure
```

DaemonSets

DaemonSets ensure that all (or some) nodes run a copy of a Pod. They are typically used for background tasks like monitoring or logging.

Example:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      name: fluentd
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      tolerations:
      - key: "node-role.kubernetes.io/master"
        effect: "NoSchedule"
      containers:
      - name: fluentd
        image: fluentd
        resources:
          requests:
            memory: "100Mi"
            cpu: "100m"
          limits:
            memory: "200Mi"
            cpu: "200m"
```

In this example, the **fluentd** DaemonSet will run a Fluentd logging agent on all nodes, including the master nodes, due to the provided toleration.

To check existing DaemonSets:

- kubectl get ds --all-namespaces

These additional features are primarily administrative tools but are crucial for effective Kubernetes cluster management. They provide fine-grained control over your cluster's behavior, resource utilization, and workload scheduling.

Kubernetes Cheatsheet

Creating Objects:-

Name	Command
Create resource	kubectl apply -f ./<file_name>.yaml
Create from multiple files	kubectl apply -f ./<file_name_1>.yaml -f ./<file_name_2>.yaml
Create all files in directory	kubectl apply -f ./<directory_name>
Create from url	kubectl apply -f https://<url>
Create pod	kubectl run <pod_name> --image <image_name>
Create pod, then expose it as service	kubectl run <pod_name> --image <image_name> --port <port> --expose
Create pod yaml file	kubectl run <pod_name> --image image_name --dry-run=client -o yaml > <file_name>.yaml
Create deployment	kubectl create deployment <deployment_name> --image <image_name>
Create deployment yaml file	kubectl create deployment <deployment_name> --image <image_name> --dry-run=client -o yaml > <file_name>.yaml
Create service	kubectl create service <service-type> <service_name> --tcp=<port:target_port>
Create service yaml file	kubectl create service <service-type> <service_name> --tcp=<port:target_port> --dry-run=client -o yaml > <file_name>.yaml
Expose service from pod/deployment	kubectl expose deployment <pod/deployment_name> --type=<service-type> --port <port> --target-port <target_port>
Create config map from key-value	kubectl create configmap <configmap_name> --from-literal=<key>:<value> --from-literal=<key>:<value>
Create config map from file	kubectl create configmap <configmap_name> --from-file=<file_name>
Create config map from env file	kubectl create configmap <configmap_name> --from-env-file=<file_name>
Create secret from key-value	kubectl create secret generic <secret_name> --from-literal=<key>:<value> --from-literal=<key>:<value>
Create secret from file	kubectl create secret generic <secret_name> --from-file=<file_name>

Name	Command
Create job	kubectl create job <job_name> --image=<image_name>
Create job from cronjob	kubectl create job <job_name> --from=cronjob/<cronjob-name>
Create cronjob	kubectl create cronjob --image=<image_name> --schedule='<cron-syntax>' -- <command> <args>

Monitoring Usage Commands:-

Name	Command
Get node cpu and memory utilization	kubectl top node <node_name>
Get pod cpu and memory utilization	kubectl top pods <pod_name>

Node Commands:-

Name	Command
Describe node	kubectl describe node <node_name>
Get node in yaml	kubectl get node <node_name> -o yaml
Get node	kubectl get node <node_name>
Drain node	kubectl drain node <node_name>
Cordon node	kubectl cordon node <node_name>
Uncordon node	kubectl uncordon node <node_name>

Pod Commands:-

Name	Command
Get pod	kubectl get pod <pod_name>
Get pod in yaml	kubectl get pod <pod_name> -o yaml
Get pod wide information	kubectl get pod <pod_name> -o wide
Get pod with watch	kubectl get pod <pod_name> -w
Edit pod	kubectl edit pod <pod_name>
Describe pod	kubectl describe pod <pod_name>
Delete pod	kubectl delete pod <pod_name>
Log pod	kubectl logs pod <pod_name>
Tail -f pod	kubectl logs pod -f <pod_name>

Name	Command
Execute into pod	kubectl exec -it pod <pod_name> /bin/bash
Running Temporary Image	kubectl run <pod_name> --image=curlimages/curl --rm -it --restart=Never -- curl <destination>

Deployment Commands:-

Name	Command
Get deployment	kubectl get deployment <deployment_name>
Get deployment in yaml	kubectl get deployment <deployment_name> -o yaml
Get deployment wide information	kubectl get deployment <deployment_name> -o wide
Edit deployment	kubectl edit deployment <deployment_name>
Describe deployment	kubectl describe deployment <deployment_name>
Delete deployment	kubectl delete deployment <deployment_name>
Log deployment	kubectl logs deployment/deployment_name -f
Update image	kubectl set image deployment <deployment_name> <container_name>=<new_image_name>
Scale deployment with replicas	kubectl scale deployment <deployment name> --replicas <replicas>

Service Commands:-

Name	Command
Get service	kubectl get service <service>
Get service in yaml	kubectl get service <service> -o yaml
Get service wide information	kubectl get service <service> -o wide
Edit service	kubectl edit service <service>
Describe service	kubectl describe service <service>
Delete service	kubectl delete service <service>

Endpoints Commands:-

Name	Command
Get endpoints	kubectl get endpoints <endpoints_name>

Ingress Commands:-

Name	Command
Get ingress	kubectl get ingress
Get ingress in yaml	kubectl get ingress -o yaml
Get ingress wide information	kubectl get ingress -o wide
Edit ingress	kubectl edit ingress <ingress_name>
Describe ingress	kubectl describe ingress <ingress_name>
Delete ingress	kubectl delete ingress <ingress_name>

DaemonSet Commands:-

Name	Command
Get daemonset	kubectl get daemonset <daemonset_name>
Get daemonset in yaml	kubectl get daemonset <daemonset_name> -o yaml
Edit daemonset	kubectl edit daemonset <daemonset_name>
Describe daemonset	kubectl describe daemonset <daemonset_name>
Delete daemonset	kubectl delete deployment <daemonset_name>

StatefulSet Commands:-

Name	Command
Get statefulset	kubectl get statefulset <statefulset_name>
Get statefulset in yaml	kubectl get statefulset <statefulset_name> -o yaml
Edit statefulset	kubectl edit statefulset <statefulset_name>
Describe statefulset	kubectl describe statefulset <statefulset_name>
Delete statefuleset	kubectl delete statefulset <statefulset_name>

ConfigMaps Commands:-

Name	Command
Get configmap	kubectl get configmap <configmap_name>
Get configmap in yaml	kubectl get configmap <configmap_name> -o yaml
Edit configmap	kubectl edit configmap <configmap_name>
Describe configmap	kubectl describe configmap <configmap_name>
Delete configmap	kubectl delete configmap <configmap_name>

Secret Commands:-

Name	Command
Get secret	kubectl get secret <secret_name>
Get secret in yaml	kubectl get secret <secret_name> -o yaml
Edit secret	kubectl edit secret <secret_name>
Describe secret	kubectl describe secret <secret_name>
Delete secret	kubectl delete secret <secret_name>

Rollout Commands:-

Name	Command
Restart deployment	kubectl rollout restart deployment <deployment_name>
Undo deployment with the latest revision	kubectl rollout undo deployment <deployment_name>
Undo deployment with specified revision	kubectl rollout undo deployment <deployment_name> --to-revision <revision_number>
Get all revisions of deployment	kubectl rollout history deployment <deployment_name>
Get specified revision of deployment	kubectl rollout history deployment <deployment_name> --revision=<revision_number>

Job Commands:-

Name	Command
Get job	kubectl get job <job_name>
Get job in yaml	kubectl get job <job_name> -o yaml
Edit job in yaml	kubectl edit job <job_name>
Describe job	kubectl describe job <job_name>
Delete job	kubectl delete job <job_name>

Cronjob Commands:-

Name	Command
Get cronjob	kubectl get cronjob cronjob_name
Get cronjob in yaml	kubectl get cronjob <cronjob_name> -o yaml

Name	Command
Edit cronjob	kubectl edit cronjob <cronjob_name>
Describe cronjob	kubectl describe cronjob <cronjob_name>
Delete cronjob	kubectl delete cronjob <cronjob_name>

Network Policy Commands:-

Name	Command
Get networkpolicy	kubectl get networkpolicy <networkpolicy_name>
Get networkpolicy in yaml	kubectl get networkpolicy <networkpolicy_name> -o yaml
Get networkpolicy wide information	kubectl get networkpolicy <networkpolicy_name> -o wide
Edit networkpolicy	kubectl edit networkpolicy <networkpolicy_name>
Describe networkpolicy	kubectl describe networkpolicy <networkpolicy_name>
Delete networkpolicy	kubectl delete networkpolicy <networkpolicy_name>

Labels and Selectors Commands:-

Name	Command
Show labels of node,pod and deployment	kubectl get <node/pod/deployment> --show-labels
Attach labels to <node/pod/deployment>	kubectl label <node/pod/deployment> <pod_name> <key>=<value>
Remove labels from <node/pod/deployment>	kubectl label <node/pod/deployment> <pod_name> <key>-
Select node,pod and deployment by using labels	kubectl get <node/pod/deployment> -l <key>=<value>