**Kubernetes (high availability, scalability, auto-healing capability)**

**(Youtube: M Prashant)**

**Kubernetes** is an open-source platform designed to automate deploying, scaling, and operating containers. It was originally developed by Google in 2014 and is now maintained by the Cloud Native Computing Foundation (CNCF).

**Key Concepts of Kubernetes**

1. **Cluster**: A set of nodes (machines) that run containerized applications managed by Kubernetes. Think of a cluster as a group of computers working together. Kubernetes uses this group to run your apps.
2. **Node**: A single machine in the Kubernetes cluster, which can be a physical or virtual machine. Each computer in the cluster is called a node. Nodes do the actual work of running your apps.
3. **Pod**: The smallest and simplest Kubernetes object. A pod is a single instance of a running process in your cluster and can contain many containers. Containers run inside a pod. All containers in a pod share the same resources.
4. **Service**: Services enable communication between different parts of your application.  A service helps your apps talk to each other and to the outside world.
5. **Deployment**: It manages the creation and scaling of pods. A deployment makes sure your app is always running the right number of copies. If one copy fails, Kubernetes will start a new one.
6. **Namespace**: A way to divide cluster resources between multiple users.

A screenshot of a computer screen

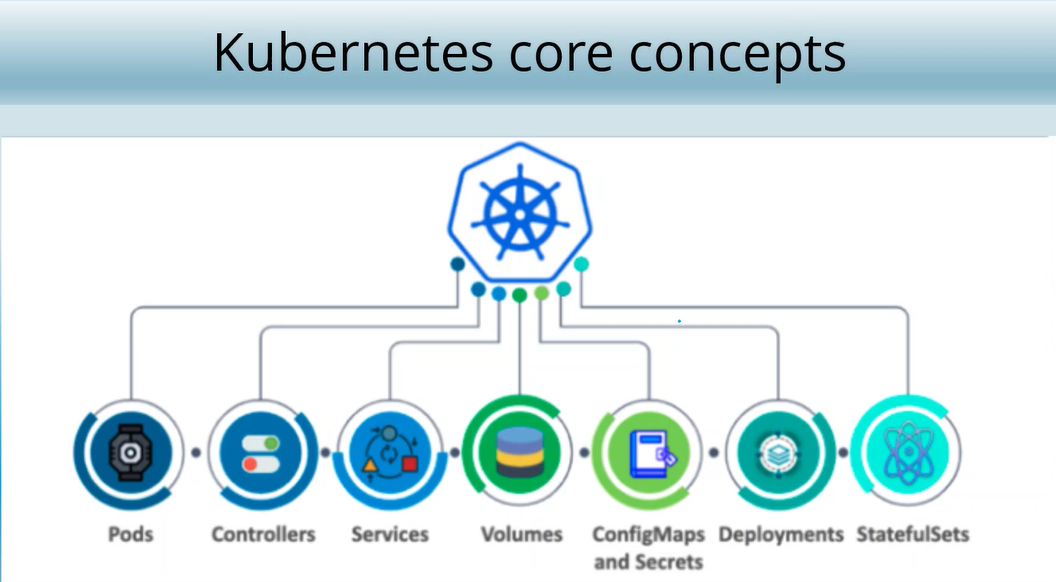
Description automatically generated

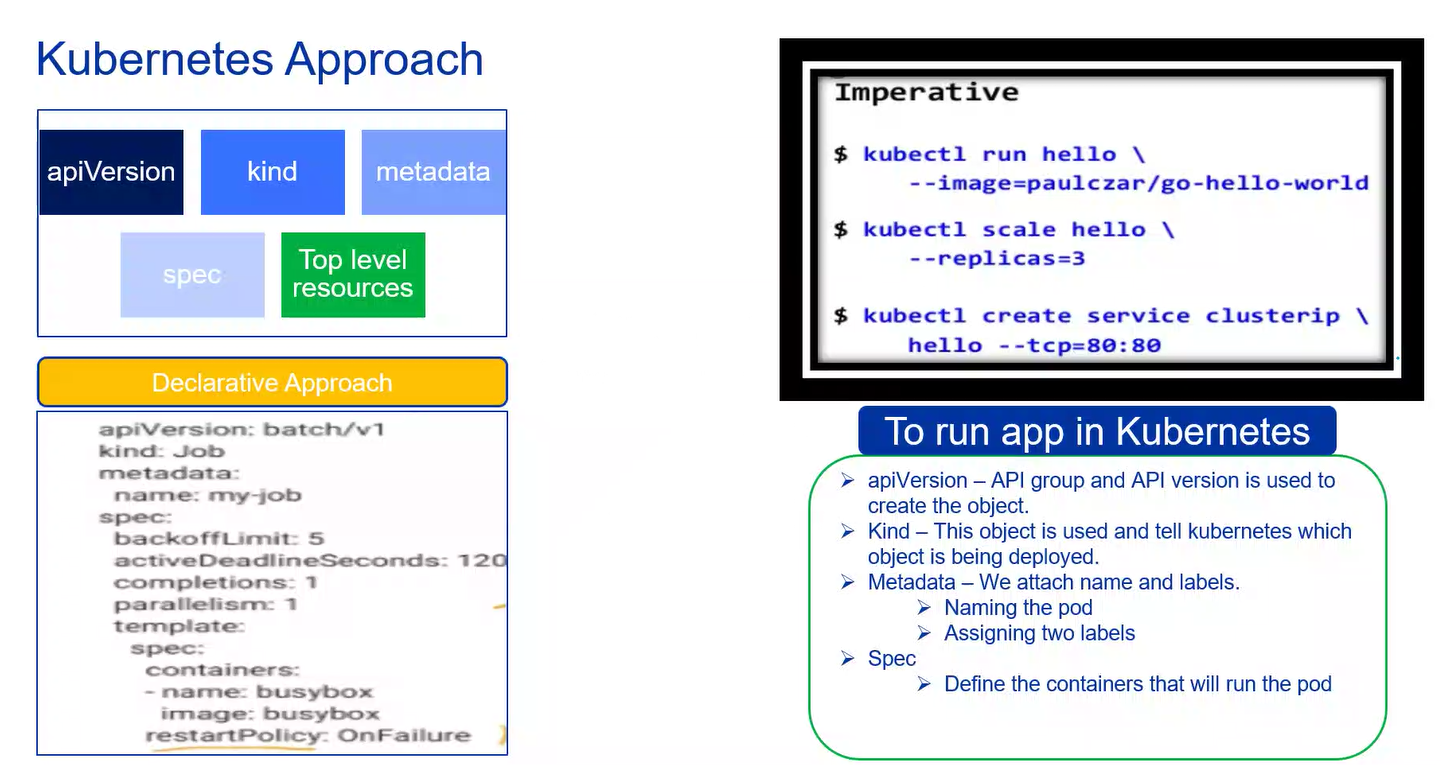
**How It Works**

1. **Deploying an App**: You tell Kubernetes what your app needs (like how many copies to run and what resources it requires). Kubernetes then makes sure your app is running as you specified.
2. **Scaling**: If your app gets popular and needs more power, you can tell Kubernetes to run more copies. It will automatically add more copies to handle the load.
3. **Self-Healing**: If something goes wrong and a part of your app crashes, Kubernetes will automatically try to fix it by restarting the failed parts.

**Example:** Imagine you have a website. You want it to be available all the time, even if one of the servers goes down. You also want to handle more visitors if your site gets popular. Kubernetes can:

* Run multiple copies of your website.
* Automatically restart any copy that crashes.
* Add more copies if you get more visitors.



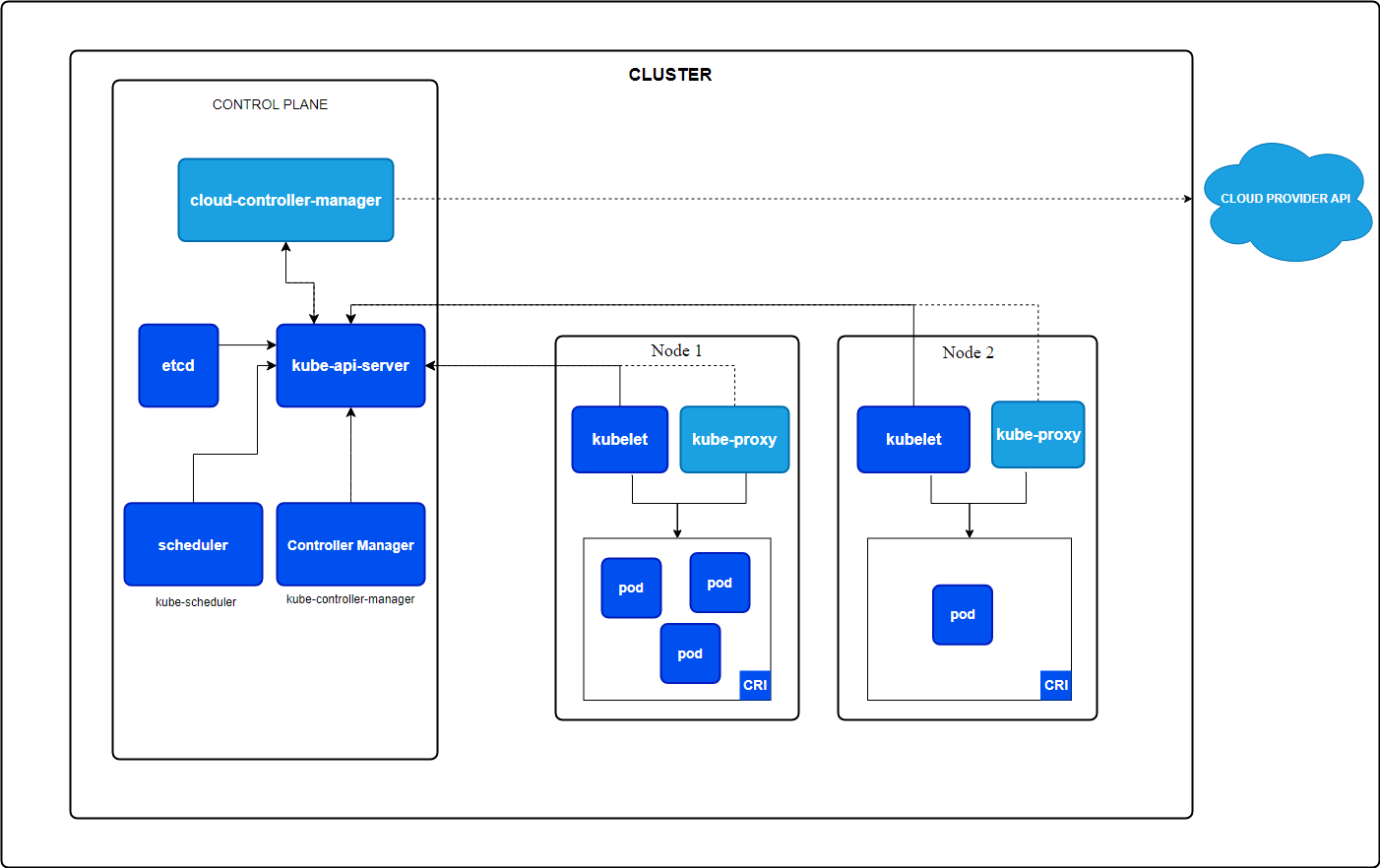


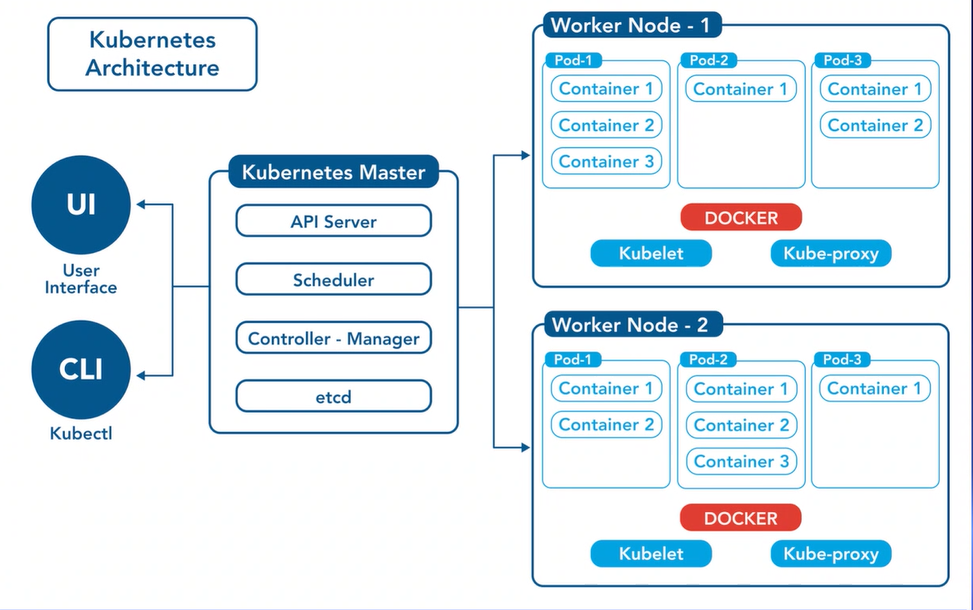
Kubernetes declarative approach and imperative approach.



Kubernetes is like a master that is used to manage containers or containerized applications just like the person is managing the instrument players

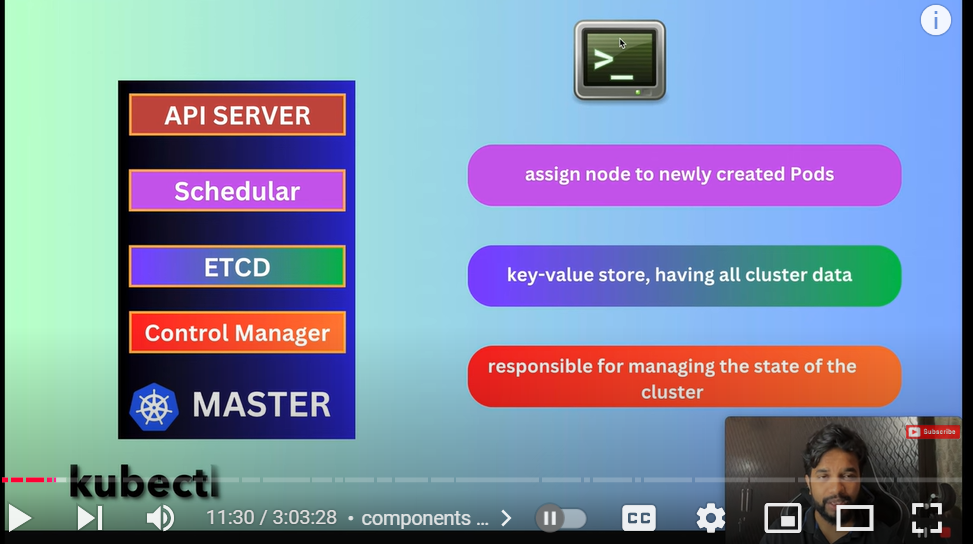
**Kubernetes Architecture**





**Kubernetes** Kubernetes uses a master-worker architecture. The master node controls and manages the cluster, while worker nodes run the applications.

**Master Components:** The master node is responsible for managing the entire Kubernetes cluster



1. **Kubectl**: Command- line interface for Kubernetes.
2. **API Server**: The API server is the front end of the Kubernetes control plane. It is used to provide a CLI called kubectl so that we can connect to the worker nodes. It receives commands from users (like creating or deleting pods) and processes them. The API Server processes the request and updates the desired state in etcd.
3. **Scheduler**: The scheduler assigns workloads to nodes. It decides which node will run a new pod based on resource availability. It’s like a traffic controller for your applications.
4. **etcd**: etcd is a distributed key-value store that stores all the cluster data.  It keeps track of the state of the cluster, such as the configuration of nodes, pods, and services. It’s like a database for Kubernetes. It is important when we are updating the Kubernetes.
5. **Controller Manager**: The controller manager is responsible for managing the state of the cluster. For example, if a pod crashes, a controller will notice and create a new one.

**Worker Components:** Worker nodes are the machines that run your applications.

A screenshot of a computer

Description automatically generated

1. **Kubelet**: The kubelet is an **agent** that runs on each worker node. Its main job is to make sure that the containers (small units of software) are running as they should be.  The kubelet gets instructions from the Kubernetes API server. These instructions describe how the containers should run.
2. **Kube-proxy**: Kube-proxy manages network rules on nodes. It handles network traffic coming into and out of the pods. It ensures that each pod can communicate with other pods and services
3. **Container Runtime**: The container runtime is the software that runs containers. (e.g., Docker). It pulls container images from a registry, starts and stops containers, and manages container storage. Docker is a common container runtime, but others like containerd and CRI-O can also be used.

**Detailed Example Workflow**

1. **Deployment**:
   * You use kubectl to deploy a web application.
   * The command is sent to the API server.
2. **API Server**: The API server processes the request and updates the desired state in etcd.
3. **Scheduler**:
   * The scheduler checks the resource availability on worker nodes.
   * It decides which worker node should run the new pod based on resource requirements and constraints.
4. **Kubelet**:
   * The kubelet on the chosen worker node receives instructions from the API server.
   * It pulls the container image using the container runtime and starts the container.
5. **Kube-proxy**: Kube-proxy sets up the necessary network rules to allow communication between the new pod and other services.
6. **Controller Manager**:
   * Continuously monitors the cluster's state.
   * If a pod fails, the replication controller ensures that a new pod is created to maintain the desired number of replicas.

**Kubernetes installation**

Initially install kubectl and install minicube

**Kubectl version –client**: This command checks whether kubectl is installled or not.

**Kubectl cluster-info**: It chekcs whether kubectl is properly configured by getting the cluster.

* **Minikube**: It is a tool used to practice Kubernetes.
* Minikube start: Start your cluster
* Minikube status: shows the state of minikube

**Kubernetes Commands**

* Minikube start: Starts the only cluster that is there in minikube.
* Minikube status: Shows the state of minikube.
* Minikube delete: Deletes the cluster
* Minikube dashboard: To open the dashboard. This dashboard shows the details and charts related to the pods and containers.
* Kubectl cluster-info: Get cluster information
* Kubectl get nodes: Get nodes
* Kubectl run <pod-name> --image= <image\_name>: To create a pod and it also creates a container
* Kubectl create deployment <deployment\_name> --image=nginx : It will create as well as deploy the pod for nginx.
* Kubectl get pods: We can see the pods with this command
* Kubectl get pods -w: We can see the pods with this command in watch mode.
* Kubectl delete pod <pod\_name>: To delete the particular pod
* Kubectl get deploy: We will get the deployment info
* Kubectl get pods -o wide: We can see the running containers inside the pod in more detailed way.
* Kubectl describe pod <pod\_name>: It shows detailed description about the particular pod.
* Kubectl expose deployment <deployment\_name> --port: <port\_number> --type: <type\_of\_service>: Expose the deployment file with port number and the service type
* Kubectl scale deployment nginx –replicas=3: Scale a deployment
* Kubectl apply -f deployment.yml: It applies means basically runs the deployment yaml file.
* Kubectl get services: Get services
* Kubectl get rs: It is used to get Replica Set. Replica Set contains the list of replicas which are mentioned in the deployment yaml file
* Kubectl create namespace <namespace\_name>: To create a namespace
* Kubectl logs <pod\_name>: Used to view logs of a particular pod
* Kubectl exec -it <pod\_name> --/bin/bash: Execute a command in a pod

**Deployment of an App in Kubernetes**

1. Create pod (container run inside pod): **kubectl create deployment <name> --image=nginx**

It creates a pod for nginx by <name>

1. **Kubectl get deployments**: give list of deployments
2. **Kubectl get Pods**: get the list of Pods
3. **Minikube dashboard**: gives you a dashboard

* **Service Object:** You directly cannot access the nginx image because it is inside the pod. For that we use:

1. Kubectl expose deployment <name> --port=80 –type= LoadBalancer

Here, <name> = name of deployment and port 80 (nginx listens on port 80)

1. Kubectl get services: gives the list of services
2. Minikube service <name> : informing minikube about out service
3. Kubectl delete deployment <name>: delete any <name> deployment

A screenshot of a computer

Description automatically generated

A diagram of a cluster of hexagons

Description automatically generated

**Rollout in Kubernetes**

A **rollout** in Kubernetes is the process of deploying a new version of your application. This involves updating the running instances (pods) of your application to a new version without causing downtime.

**Rollout Process**

1. **Create a Deployment**: You start by creating a Deployment that specifies the desired state of your application, including the number of replicas (pods) and the container image to use.

**Command**: kubectl create -f <deployment-file>.yaml

1. **Update the Deployment**: When you want to deploy a new version of your application, you update the Deployment with the new container image. Kubernetes then starts the rollout process.

**Command**: kubectl set image deployment/<deployment-name> <container-name>=<new-image>

1. **Rolling Update**: Kubernetes performs a rolling update, which means it gradually replaces the old pods with new ones. This ensures that your application remains available during the update. It updates a few pods at a time, waits for them to become ready, and then proceeds with the next set.

**Command**: kubectl rollout status deployment/<deployment-name>

1. **Rollback**: If something goes wrong with the new version, you can roll back to the previous version using the kubectl rollout undo command. This reverts the Deployment to the last known good state.

**Command**: kubectl rollout undo deployment/<deployment-name>

**Self- Healing in Kubernetes**

So if by chance your website goes down due to a glitch, then the self healing comes into action.

**How Self-Healing Works**

1. **Pod Monitoring**: Kubernetes continuously monitors the health of pods. If a pod fails (e.g., crashes or becomes unresponsive), Kubernetes will try to restart it.
2. **ReplicaSets**: These ensure that a specified number of pod replicas are running at all times. If a pod goes down, the ReplicaSet controller will create a new one to replace it.
3. **Liveness and Readiness Probes**: These are checks that Kubernetes performs to determine if a pod is healthy and ready to serve traffic.
   * **Liveness Probe**: Checks if the pod is alive. If it fails, Kubernetes will restart the pod.
   * **Readiness Probe**: Checks if the pod is ready to handle requests. If it fails, the pod is removed from the service’s endpoints until it becomes ready again.

**Scale up of application in Kubernetes**

Scaling up means adding more pods to it, which will make sure that your application will never go down because during self healing it goes down for few seconds, but scle up will never let your website go down.

**Command:** kubectl scale deployment <dep\_name> --replicas = 10

Make 10 replicas of that deployment.

**YAML configuration for deployment and service**

**Deployment File:** A **Deployment** in Kubernetes is used to manage a set of identical pods. It ensures that the desired number of pods are running and can update them to a new version without downtime.

**Key Components of a Deployment File**

1. **apiVersion**: Specifies the API version, e.g., apps/v1.
2. **kind**: Specifies the type of Kubernetes resource. Eg: Pod ReplicaSet,

Deployment, Service.

1. **metadata**: Provides metadata for the Deployment, including the name and labels. This is a dictionary (key-value pairs).
2. **spec**: **It is the most crucial part of the file.** Defines the desired state of the Deployment, including:
   * **replicas**: Number of pod replicas to run.
   * **selector**: Labels used to select the pods managed by this Deployment.
   * **template**: Defines the pod template used for creating new pods, including container specifications.

**Example Deployment File**

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app

spec:

replicas: 3

selector:

matchLabels:

app: my-app

template:

metadata:

labels:

app: my-app

spec:

containers:

- name: my-app

image: my-app:1.0

ports:

- containerPort: 80

**Service File:** A **Service** in Kubernetes is used to expose your pods to the network, making them accessible to other services or users. It provides a stable IP address and DNS name for a set of pods.

**Key Components of a Service File**

1. **apiVersion**: Specifies the API version, e.g., v1.
2. **kind**: Specifies the type of Kubernetes resource, in this case, Service.
3. **metadata**: Provides metadata for the Service, including the name and labels.
4. **spec**: Defines the desired state of the Service, including:
   * **selector**: Labels used to select the pods that the Service will expose.
   * **ports**: Specifies the ports that the Service will expose.
   * **type**: Defines the type of Service (e.g., ClusterIP, NodePort, LoadBalancer).

**Example Service File**

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

selector:

app: my-app

ports:

- protocol: TCP

port: 80

targetPort: 80

type: ClusterIP

**Deploying Multiple Containers in a Single Pod**

**Why Use Multiple Containers in a Single Pod?**

1. **Tight Coupling**: When containers are tightly coupled and need to share resources like storage or network.
2. **Sidecar Pattern**: One container might handle the main application, while another handles auxiliary tasks like logging, monitoring, or proxying (e.g., a web server and a logging agent).
3. **Shared Lifecycle**: Containers in the same pod share the same lifecycle. They start, stop, and restart together.

**How It Works**

* **Shared Storage**: Containers in a pod can share volumes, allowing them to read and write to the same files.
* **Shared Network**: They share the same IP address and port space, making inter-container communication straightforward.
* **Configuration**: You define multiple containers in the same pod specification in your YAML file.

**Deploying Multiple Containers in Separate Pods**

**Why Use Separate Pods?**

1. **Scalability**: You can scale each container independently. For example, if your frontend needs more instances than your backend, you can scale them separately.
2. **Isolation**: Each pod is isolated, which can improve security and fault tolerance. If one pod fails, it doesn’t directly affect the others.
3. **Flexibility**: Different containers can be scheduled on different nodes, optimizing resource usage across the cluster.

**How It Works**

* **Independent Scaling**: Each pod can be scaled independently using Kubernetes deployments.
* **Service Discovery**: Kubernetes services can be used to enable communication between pods. Each pod gets its own IP address.
* **Configuration**: You define each pod separately in your YAML files.

**ConfigMaps**

**What is a ConfigMap?**

A **ConfigMap** is a Kubernetes object used to store configuration data in key-value pairs. This data can then be injected into your pods, making it easier to manage configuration separately from the pod definitions.

**Why Use ConfigMaps?**

When you have many pod definition files, managing environment variables directly within these files can become difficult. ConfigMaps centralize this configuration data, making it easier to manage and update.

**How to Create a ConfigMap**

You can create a ConfigMap by defining it in a YAML file.

apiVersion: v1

kind: ConfigMap

metadata:

name: my-config

data:

key1: value1

key2: value2

**Using ConfigMaps in Pods**

There are several ways to use ConfigMaps in your pods:

1. **Environment Variables**: Inject configuration data as environment variables.
2. **Command-Line Arguments**: Pass configuration data as arguments to the container’s command.
3. **Configuration Files**: Mount the ConfigMap as a volume and use it as a configuration file.

**Working with Secrets in Kubernetes**

In Kubernetes, **Secrets** are used to store sensitive information like passwords, tokens, or keys. They are similar to ConfigMaps, but they store data in an encoded format for added security. Let's break down how to create and use secrets in detail and simple language.

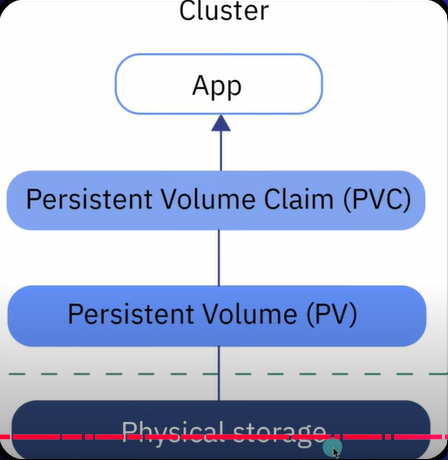
**Why Use Secrets?**

Hardcoding sensitive information like database passwords directly into your application code or Kubernetes Pod definition files is not secure. Instead, you can use Kubernetes Secrets to manage this data securely and inject it into your pods.

**Creating Secrets**

There are two main ways to create Secrets in Kubernetes: the imperative way (command line) and the declarative way (using a YAML file).

**Volumes and Data**



**What is a Volume in Kubernetes?**

In Kubernetes, a **volume** is essentially a directory that contains data, which is accessible to the containers in a pod. Unlike the data inside a container, which is ephemeral (temporary), data in a volume can persist across container restarts and crashes.

**Why Use Volumes?**

1. **Persistence**: Data in a volume remains available even if the container using it crashes or restarts.
2. **Sharing**: Multiple containers within the same pod can access the same data

**Types of Volumes:** Kubernetes supports several types of volumes, each suited for different use cases:

1. **emptyDir**: An empty folder that is created when the pod starts and is deleted when the pod stops. Good for temporary storage.
2. **hostPath**: Uses a folder from the host machine where the pod is running. Useful for accessing host files.
3. **configMap**: Provides a way to inject configuration data into pods.  Stores configuration data that can be used by the containers.
4. **persistentVolume (PV)**: A piece of storage in the cluster that is set up by an admin. It’s not tied to any specific pod.
5. **persistentVolumeClaim (PVC)**: A request for storage by a user. It can automatically get the storage it needs

**How to Use Volumes:** To use a volume in a pod, you need to:

1. **Define the Volume**: Specify the volume in the pod’s configuration.
2. **Mount the Volume**: Specify where the volume should be mounted within the container.

**Persistent Volumes and Claims:** For more durable storage, you can use **Persistent Volumes (PVs)** and **Persistent Volume Claims (PVCs)**:

1. **Persistent Volume (PV)**: A cluster-wide resource that is provisioned by an administrator. It represents actual storage.
2. **Persistent Volume Claim (PVC)**: A request for storage by a user It finds a PV that matches the request

**Example of a PVC**:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mypvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

In this example: We create a PVC named mypvc requesting 1Gi of storage with ReadWriteOnce access mode.

**Kubernetes Udemy**

**Container Runtime Interface**

**What is CRI?**

The **Container Runtime Interface (CRI)** is a standard way for Kubernetes to interact with different container runtimes. Think of it as a translator that allows Kubernetes to work with various container tools without needing to change its own code.

**Why CRI is Important**

Before CRI, Kubernetes was tightly coupled with Docker. This meant Kubernetes could only work with Docker containers. As other container runtimes like containerd and CRI-O emerged, there was a need for Kubernetes to support these new tools. CRI was introduced to solve this problem.

**How CRI Works**

1. **Kubelet and CRI**: The kubelet is a component of Kubernetes that runs on each node and is responsible for managing the containers. CRI allows the kubelet to communicate with any container runtime that implements the CRI standard.
2. **gRPC Protocol**: CRI uses a protocol called gRPC (Google Remote Procedure Call) to communicate between the kubelet and the container runtime. [This ensures efficient and reliable communication1](https://kubernetes.io/docs/concepts/architecture/cri/).

**Components of CRI:** CRI defines two main services:

1. **Runtime Service**: Manages the lifecycle of pods and containers (e.g., creating, starting, stopping containers).
2. **Image Service**: Manages container images (e.g., pulling, listing, and removing images).

* **Dockershim**: Docker was not originally built to support CRI. To keep supporting Docker, Kubernetes introduced dockershim, a temporary solution that allowed Docker to work with Kubernetes outside of the CRI framework.
* **Removal of Dockershim**: Maintaining dockershim added complexity, so Kubernetes decided to remove it in version 1.24. This means Kubernetes no longer supports Docker as a runtime directly, but Docker images still work because they follow the OCI ImageSpec.

**CLI Tools for Managing Containers**

1. **nerdctl**

* **Purpose**: A Docker-like CLI for Containerd.
* **Usage**: Provides a user-friendly interface similar to Docker CLI, with additional features.
* **Example**:

nerdctl pull redis

nerdctl run -d --name redis redis:latest

* **Features**: Supports encrypted container images, lazy pulling, P2P image distribution, and more.

1. **crictl**

* **Purpose**: A CLI tool for interacting with CRI-compatible container runtimes.
* **Usage**: Useful for Kubernetes administrators to manage pods and containers.
* **Example**:

crictl pull redis

crictl runp pod-config.json

* **Features**: Can list pods, containers, and images, and perform operations like viewing logs and executing commands inside containers.

**Summary of CLI Tools**

* **ctr**: Used for debugging Containerd, limited features.
* **nerdctl**: Docker-like CLI for Containerd, user-friendly, supports advanced features.
* **crictl**: Developed by the Kubernetes community, works with all CRI-compatible runtimes, mainly for debugging and inspecting.

**ETCD**

**What is ETCD?**

**ETCD** is a distributed, reliable key-value store. It is used to store configuration data, state data, and metadata for Kubernetes. **It acts as the primary data store for all cluster information.**

**Client Tool (ETCDCTL)**: ETCD comes with a command-line tool called ETCDCTL. You can use it to store and retrieve key-value pairs.

**Store Data**: etcdctl put key1 value1

**Retrieve Data**: etcdctl get key1

**What ETCD Stores:** ETCD stores all the configuration data and state information for a Kubernetes cluster. This includes:

* **Nodes**: Information about each node in the cluster.
* **Pods**: Details about all the pods running in the cluster.
* **Configs**: Configuration data for various Kubernetes resources.
* **Secrets**: Sensitive information like passwords and keys.
* **Accounts and Roles**: User accounts and their permissions.
* **Role Bindings**: Links between roles and users or groups.

**How ETCD Works in Kubernetes:** When you interact with your Kubernetes cluster using commands like kubectl get, the information you see is retrieved from ETCD. Similarly, any changes you make, such as adding nodes or deploying pods, are first recorded in ETCD. Only after ETCD is updated are these changes considered complete.

**Deployment:** To deploy ETCD in a Kubernetes cluster, we use kubeadm

**Using kubeadm**:

* + **Automated Setup**: kubeadm handles the deployment of ETCD for you.
  + **ETCD as a Pod**: ETCD runs as a pod in the kube-system namespace.
  + **Exploring ETCD**: You can use the etcdctl command-line tool within this pod to interact with the ETCD database.

**Example Commands**

* **List All Keys**: etcdctl get / --prefix --keys-only
* **Set a Key-Value Pair**: etcdctl put mykey "Hello ETCD"
* **Get a Key-Value Pair**: planeetcdctl get mykey

**API Server in Kubernetes**

It acts as the front end for the Kubernetes master, handling all the communication between the various components of the cluster.

**Key Functions of kube-apiserver**

1. **Authentication and Validation**:
   * When you run a kubectl command, it sends a request to the kube-apiserver.
   * The kube-apiserver first authenticates the request to ensure it comes from a legitimate source.
   * It then validates the request to check if it is correctly formatted and authorized.
2. **Data Retrieval and Updates**:
   * The kube-apiserver retrieves data from the **etcd** cluster, which is the primary data store for Kubernetes.
   * It responds back with the requested information after fetching it from etcd.
3. **Direct API Access**:
   * You can interact with the kube-apiserver directly using API requests, not just through kubectl.
   * For example, you can create a pod by sending a POST request to the API server.

**Example Workflow: Creating a Pod**

1. **Request Handling**:
   * You send a request to create a pod using kubectl or directly via the API.
   * The kube-apiserver authenticates and validates this request.
2. **Pod Creation**:
   * The kube-apiserver creates a pod object but does not assign it to a node yet.
   * It updates this information in the etcd server.
3. **Scheduler's Role**:
   * The scheduler monitors the kube-apiserver and notices the new pod without a node assignment.
   * It selects an appropriate node for the pod and informs the kube-apiserver.
4. **Node Assignment**:
   * The kube-apiserver updates the etcd server with the node assignment.
   * It then informs the kubelet on the selected node.
5. **Pod Deployment**:
   * The kubelet on the node creates the pod and instructs the container runtime to deploy the application image.
   * Once the pod is running, the kubelet updates the status back to the kube-apiserver.
   * The kube-apiserver updates the etcd server with the final status.

**Kube Controller Manager**

The **Kube Controller Manager** is a key component in Kubernetes that manages various controllers. These controllers are responsible for ensuring that the desired state of the cluster matches the actual state.

**What is a Controller?**

A **controller** in Kubernetes is a process that continuously monitors the state of different components in the cluster and makes adjustments to ensure everything is running as expected. Think of controllers as specialized departments within an organization, each with its own responsibilities.

**Key Controllers Managed by Kube Controller Manager**

1. **Node Controller**:
   * **Purpose**: Monitors the health of nodes.
   * **Function**: Checks the status of nodes every few seconds. If a node stops sending heartbeats, it is marked as unreachable after 40 seconds. If the node remains unreachable for five minutes, the controller removes the pods assigned to that node and reschedules them on healthy nodes.
2. **Replication Controller**:
   * **Purpose**: Ensures that the desired number of pod replicas are running.
   * **Function**: Monitors the status of replica sets. If a pod dies, it creates a new one to maintain the desired number of replicas.

**Scheduler**

The **kube-scheduler** is a crucial component in Kubernetes responsible for deciding which pods go on which nodes. It doesn't actually place the pods on the nodes; that job is handled by the kubelet. The scheduler's role is to determine the best node for each pod based on various criteria.

**Why Do We Need a Scheduler?**

In a Kubernetes cluster, there are many nodes (like ships) and many pods (like containers). The scheduler ensures that each pod is placed on the most suitable node. This involves considering factors like resource availability and specific requirements of the pods.

**How the Scheduler Works**

1. **Filtering Phase**: The scheduler first filters out nodes that do not meet the resource requirements of the pod. For example, if a pod requires a certain amount of CPU and memory, nodes that do not have enough resources are excluded.
2. **Scoring Phase**: After filtering, the scheduler ranks the remaining nodes to find the best fit for the pod. It uses a priority function to assign a score to each node based on various factors, such as the amount of free resources after placing the pod.

**Example Scenario:** Imagine you have a pod that requires a specific amount of CPU and memory. The scheduler will:

1. **Filter Nodes**: Exclude nodes that do not have enough CPU and memory.
2. **Rank Nodes**: Assign scores to the remaining nodes based on how well they can accommodate the pod. The node with the highest score is selected.

**Kubelet**

The **kubelet** is a crucial component in Kubernetes, acting as the primary "node agent" that runs on each node (worker machine) in the cluster. It ensures that containers are running and healthy on the node.

**Key Responsibilities of kubelet**

1. **Node Registration**: The kubelet registers the node with the Kubernetes cluster. This involves sending information about the node to the Kubernetes API server.
2. **Pod Management**: When the kubelet receives instructions to run a pod, it communicates with the container runtime (like Docker or containerd) to pull the required container images and start the containers.
3. **Health Reporting**: The kubelet sends regular updates (heartbeats) to the API server about the status of the node and the containers running on it. This helps Kubernetes keep track of the health and status of all nodes in the cluster.

**How kubelet Works**

1. **Receiving Instructions**: The kubelet receives instructions from the Kubernetes API server, which may include commands to start or stop pods.
2. **Interacting with Container Runtime**: The kubelet interacts with the container runtime to manage the lifecycle of containers. For example, it might instruct Docker to pull an image and run a container.
3. **Monitoring and Reporting**: The kubelet monitors the health and status of the containers and pods running on the node. It reports this information back to the API server, ensuring that the cluster's state is accurately reflected.

**Kube-proxy**

**kube-proxy** is a critical component in Kubernetes that manages network communication within the cluster. It ensures that each pod can communicate with other pods, even if they are on different nodes.

**Pod Networking:** In a Kubernetes cluster, every pod can reach every other pod through a **pod network**. This is an internal virtual network that spans across all nodes in the cluster

**Services and kube-proxy:** When you deploy applications in Kubernetes, you often use **Services** to expose your applications. A Service provides a stable IP address and DNS name to access a set of pods.

**Example Scenario**

* **Web Application and Database**: Suppose you have a web application running on one node and a database running on another node. The web app can reach the database using the pod's IP address, but this IP might change if the pod is recreated.
* **Using a Service**: To solve this, you create a Service for the database. The web application can now access the database using the Service's name (e.g., db-service) or its stable IP address.

**How kube-proxy Works: kube-proxy** runs on each node in the Kubernetes cluster. Its main job is to manage network rules and ensure that traffic is correctly routed to the appropriate pods.

1. **Service Creation**: When a new Service is created, kube-proxy sets up the necessary rules to forward traffic to the backend pods. It uses **iptables** rules to achieve this.
2. **Traffic Routing**: For example, if a Service has an IP address of 10.96.0.12 and the backend pod has an IP address of 10.32.0.15, kube-proxy creates an iptables rule to forward traffic from 10.96.0.12 to 10.32.0.15.

**Replication Controller and Replication Set**

**What is a Replication Controller?**

A Replication Controller ensures that a specified number of pod replicas are running at any given time. If a pod fails or is deleted, the Replication Controller will create a new one to replace it, ensuring high availability and reliability of your application.

**Replication Controller vs. Replica Set**

* **Replication Controller**: This is the older technology used to manage pod replicas.
* **Replica Set**: This is the newer, recommended way to manage pod replicas. It offers more features and is more efficient.

**Key Features of ReplicaSet**

* **Label Selectors**: Uses label selectors to identify the pods it should manage. This allows for flexible and dynamic management of pods.
* **Declarative Management**: You declare the desired state (number of replicas) in a YAML file, and the ReplicaSet ensures that this state is maintained.
* **Self-Healing**: Automatically replaces failed or deleted pods to maintain the desired number of replicas.

**Services**

**What are Kubernetes Services?**

Kubernetes Services are a way to enable communication between various components within a Kubernetes cluster and with the outside world. They help connect applications together, allowing different parts of your application to communicate with each other and with external users.

**Types of Services**

1. **ClusterIP**:
   * **Default type**.
   * Exposes the service on an internal IP in the cluster.
   * Makes the service accessible only within the cluster.
   * Useful for communication between different services within the cluster.
2. **NodePort**:
   * Exposes the service on a static port on each node’s IP.
   * Makes the service accessible from outside the cluster using <NodeIP>:<NodePort>.
   * Useful for simple, external access to services.
   * Important ports in NodePort

* **TargetPort:** The port on the pod where the application is running (e.g., 80).
* **Port:** The port on the service itself (e.g., 80).
* **NodePort:** The port on the node that maps to the service (e.g., 30008).

When you access http://<NodeIP>:30008 from your laptop, the request is forwarded to the service, which then forwards it to the pod on port 80.

1. **LoadBalancer**:
   * Exposes the service externally using a cloud provider’s load balancer.
   * Distributes traffic across multiple nodes.
   * Useful for production environments where you need to balance load across multiple instances.

**Namespace**

Namespaces in Kubernetes are like separate environments within a single Kubernetes cluster. They help organize and manage resources, especially in large clusters or when multiple teams or projects share the same cluster.

**Why Do We Need Namespaces?**

Namespaces provide a way to divide cluster resources between multiple users (via resource quotas). They also help in managing and organizing resources, ensuring that different environments (like development, testing, and production) can coexist without interfering with each other.

**Analogy to Understand Namespaces:** Think of namespaces like houses in a neighborhood:

* Each house (namespace) has its own set of rules and resources.
* People inside a house can refer to each other by their first names.
* To refer to someone in another house, you use their full name (first name + last name).

Similarly, in Kubernetes:

* Each namespace has its own set of resources and policies.
* Resources within a namespace can refer to each other by their names.
* To refer to a resource in another namespace, you use the full name (resource name + namespace).

**Default Namespaces in Kubernetes**

1. **default**: The default namespace where user-created resources are placed if no other namespace is specified.
2. **kube-system**: Contains Kubernetes system components like the API server, scheduler, and controllers.
3. **kube-public**: A special namespace that is readable by all users (including those not authenticated). It is mostly used for public resources.
4. **Kube**-node-lease: It stores the lease objects.

**Kubectl Apply**

**What is**kubectl apply**?**

The kubectl apply command is used to manage Kubernetes objects in a declarative way. This means you define the desired state of your objects in configuration files (usually YAML), and kubectl apply ensures that the actual state of the cluster matches this desired state.

**How Does**kubectl apply**Work?**

When you run the kubectl apply command, it considers three main configurations:

1. **Local Configuration File**: The YAML file you have on your local system.
2. **Live Configuration**: The current state of the object in the Kubernetes cluster.
3. **Last Applied Configuration**: The last configuration that was applied to the object

**Labels, Selectors and Annotations**

* **Labels in Kubernetes: Labels** are key-value pairs that are attached to Kubernetes objects such as pods, services, and deployments. They are used to organize and categorize these objects.

**Example:**

metadata:

labels:

app: my-app

environment: production

tier: frontend

In this example, the labels app, environment, and tier are attached to an object, helping to categorize it as part of the my-app application, in the production environment, and belonging to the frontend tier.

* **Selectors in Kubernetes: Selectors** are used to filter and select Kubernetes objects based on their labels. They allow you to group objects and perform operations on them collectively.

**Example:**

To select all pods with the label app=my-app:

kubectl get pods -l app=my-app

* **Annotations in Kubernetes: Annotations** are also key-value pairs, but unlike labels, they are not used to identify and select objects. Instead, annotations are used to store additional metadata about objects, such as build information, contact details, or configuration data.

**Example:**

metadata:

annotations:

buildVersion: "1.0.0"

maintainer: "team@example.com"

Annotations provide a way to attach arbitrary non-identifying metadata to objects.

**Taints and Tolerations?**

**Taints** and **tolerations** are mechanisms in Kubernetes that control which pods can be scheduled on which nodes. They help ensure that certain pods are only placed on specific nodes, or that certain nodes are reserved for specific types of workloads.

**Analogy to Understand Taints and Tolerations:** Imagine a person sprayed with a bug repellent (odomos) (taint). Bugs (pods) that are intolerant to the repellent will avoid landing on the person. However, some bugs might be tolerant to the repellent and can land on the person despite the spray.

* **Person**: Node
* **Bug**: Pod
* **Repellent**: Taint
* **Tolerance to Repellent**: Toleration

**How Taints and Tolerations Work**

1. **Taints**: Applied to nodes to repel certain pods.
2. **Tolerations**: Applied to pods to allow them to tolerate specific taints and be scheduled on nodes with those taints.

**Example Scenario**

**Initial Setup**

* **Nodes**: Three worker nodes named node1, node2, and node3.
* **Pods**: Four pods named A, B, C, and D.

Initially, the Kubernetes scheduler places the pods across the nodes to balance the load.

**Taint Effects**

There are three types of taint effects:

1. **NoSchedule**: Pods that do not tolerate the taint will not be scheduled on the node.
2. **PreferNoSchedule**: The system will try to avoid placing pods that do not tolerate the taint on the node, but it is not guaranteed.
3. **NoExecute**: Pods that do not tolerate the taint will not be scheduled on the node, and existing pods on the node will be evicted.

**Node Selectors**

**Node selectors** are a simple way to constrain pods to only be scheduled on nodes that meet certain criteria. This is done by adding labels to nodes and then specifying those labels in the pod’s configuration.

**Example Scenario**

Imagine you have a Kubernetes cluster with three nodes:

* **Node1**: A large node with high resources.
* **Node2**: A smaller node with lower resources.
* **Node3**: Another smaller node with lower resources.

You want to ensure that a data processing workload, which requires high resources, is always scheduled on the large node (Node1).

**Node Affinity: Node affinity** is a more flexible and powerful way to control which nodes your pods can be scheduled on. It allows you to use more complex rules and expressions.

**Example Scenario**

Continuing with the previous example, let’s say you want to place the pod on nodes that are either large or medium, or avoid small nodes.

**Steps to Use Node Affinity**

1. **Label the Nodes**: Label the nodes as needed:

kubectl label nodes node1 size=large

1. Add a **nodeAffinity** field to the pod’s YAML file

**Types of Node Affinity**

1. **requiredDuringSchedulingIgnoredDuringExecution**:
   * The pod must be scheduled on a node that matches the affinity rules.
   * If no matching node is found, the pod will not be scheduled.
   * Once scheduled, the pod will not be evicted if the node’s labels change.
2. **preferredDuringSchedulingIgnoredDuringExecution**:
   * The scheduler will try to place the pod on a node that matches the affinity rules, but it’s not mandatory.
   * If no matching node is found, the pod can be scheduled on any available node.
   * Once scheduled, the pod will not be evicted if the node’s labels change.

**Static Pods**

**Static Pods** are pods that are managed directly by the kubelet on a specific node, without the need for the kube-apiserver, kube-scheduler, or controllers. **How Static Pods Work**

1. **No Control Plane Needed**: Static Pods do not rely on the Kubernetes control plane. This means they can run even if there is no kube-apiserver, kube-scheduler, or ETCD cluster.
2. **Managed by Kubelet**: The kubelet on each node is responsible for managing static Pods. It reads the Pod definition files from a specified directory on the node.
3. **Pod Definition Files**: You provide the kubelet with Pod definition files, which it uses to create and manage the Pods.

Kubernetes Scheduler working

**How the Kubernetes Scheduler Works**

When you create a pod in Kubernetes, it needs to be scheduled onto a node. The scheduler is responsible for this task.

**1. Scheduling Queue:** When a pod is created, it enters the scheduling queue. Pods in this queue are sorted based on their priority.

* **Priority Classes**: You can define priority classes to assign different priorities to pods. Higher priority pods are scheduled first.

**2. Filter Phase:** In this phase, the scheduler filters out nodes that cannot run the pod. This is based on the pod’s resource requirements and other constraints.

* **Example**: If a pod requires 10 CPU units, nodes with less than 10 CPU units available are filtered out.

**3. Scoring Phase:** Nodes that pass the filter phase are then scored. The scheduler assigns scores to nodes based on various criteria to determine the best node for the pod.

* **Example**: Nodes are scored based on the remaining resources after the pod is scheduled. Nodes with more remaining resources get higher scores.

**4. Binding Phase:** Finally, the pod is bound to the node with the highest score. This means the pod is assigned to that node and starts running there.

**Section 4: Logging and Monitoring**

**Monitoring a Kubernetes Cluster**

Monitoring a Kubernetes cluster involves tracking various metrics to ensure that your nodes and pods are running smoothly. You want to monitor:

1. **Node-Level Metrics**:
   * Number of nodes
   * Health status of nodes
   * CPU, memory, network, and disk utilization
2. **Pod-Level Metrics**:
   * Number of pods
   * CPU and memory usage for each pod

To achieve this, you need a solution that can collect, store, and analyze these metrics.

**Available Monitoring Solutions:** Kubernetes does not come with a full-featured built-in monitoring solution, but several open-source and proprietary options are available:

1. **Metrics Server**: A lightweight monitoring solution that collects metrics from nodes and pods. It stores data in memory and does not keep historical performance data.
2. **Prometheus**: A powerful open-source monitoring and alerting toolkit that provides a robust way to store metrics and historical data.
3. **Elastic Stack**: A collection of open-source tools (Elasticsearch, Logstash, Kibana, and Beats) for searching, analyzing, and visualizing data.
4. **Proprietary Solutions**: Tools like Datadog and Dynatrace offer comprehensive monitoring solutions with advanced features.

**Section 5: Application Lifecycle Management**

**Multi-Container Pods**

A **multi-container pod** is a pod that runs more than one container. These containers share the same network and storage, and they can communicate with each other using localhost.

**Why Use Multi-Container Pods?**

Multi-container pods are useful when you need multiple processes to work together closely. Here are some scenarios where multi-container pods are beneficial:

1. **Sidecar Containers**: Containers that assist the main container by providing additional functionality, such as logging, monitoring, or proxying requests.
2. **Adapter Containers**: Containers that modify the output of one container to fit the input requirements of another.
3. **Ambassador Containers**: Containers that proxy network traffic to other services.

**Structure of a Multi-Container Pod:** In a multi-container pod, containers share the following resources:

* **Network**: All containers in the pod share the same IP address and port space.
* **Storage**: Containers can share data through volumes mounted to the pod.

**Init Containers**

**Init containers** are special containers that run before the main application containers in a pod. They are used to perform initialization tasks that must complete before the main containers start.

**Why Use Init Containers?**

Init containers are useful for setting up the environment for the main application containers. They help in performing tasks like:

* Setting up configuration
* Ensuring dependencies are met
* Initializing databases
* Fetching secrets or configurations

**Key Features of Init Containers**

1. **Run Sequentially**: Init containers run one after another in a specific order. Each init container must complete successfully before the next one starts.
2. **Block Main Containers**: The main application containers do not start until all init containers have completed.
3. **Isolated from Main Containers**: Init containers can have different images and settings than the main containers. They are separate and can use different tools or libraries.
4. **Retry on Failure**: If an init container fails, Kubernetes retries it until it succeeds. This ensures that the initialization tasks are completed before the main application starts.

**Example Use Case: Web Application with Database Initialization**

Imagine you have a web application that relies on a database. Before the web server starts, you need to:

1. Ensure the database is up and running.
2. Load initial data into the database.

**Upgrading a Kubernetes Cluster**

Upgrading a Kubernetes cluster is crucial to ensure you're using the latest features, performance improvements, and security patches

**Understanding Component Versions:** In Kubernetes, different components can be at different versions. It's important to ensure compatibility:

* **kube-apiserver**: The primary control plane component. All other components interact with it and should never be at a higher version. Example: Version 1.10
* **controller-manager and scheduler**: Can be one version lower than kube-apiserver. Example: Version 1.10 or 1.9
* **kubelet and kube-proxy**: Can be up to two versions lower than kube-apiserver. Example: Version 1.10, 1.9, or 1.8.
* **kubectl**: Can be at a version higher, the same, or one version lower than kube-apiserver. Example: Version 1.11, 1.10, 1.9

**Version Compatibility Example**

If kube-apiserver is at version 1.10:

* Controller-manager and scheduler can be 1.10 or 1.9.
* Kubelet and kube-proxy can be 1.10, 1.9, or 1.8.
* kubectl can be 1.11, 1.10, or 1.9.

**Upgrade Strategies**

1. **Managed Kubernetes Services**: Cloud providers like Google Kubernetes Engine offer easy upgrade options through their dashboards.
2. **kubeadm Tool**: Helps plan and upgrade clusters with commands.
3. **Manual Upgrades**: For clusters set up from scratch, you manually upgrade each component.

.

**Backup and Restore in Kubernetes**

In Kubernetes, ensuring you have proper backup and restore in place is crucial for maintaining the reliability and availability of your applications.

**What to Backup in a Kubernetes Cluster**

1. **Resource Configurations**: These include deployments, pods, services, ConfigMaps, secrets, and other objects.
2. **Persistent Storage**: If your applications use persistent volumes, the data stored in these volumes should be backed up.
3. **etcd Database**: etcd stores the state of the entire cluster, including resource configurations and metadata.

**Backup Strategies**

1. **Resource Configurations**

**Declarative Approach:** This involves creating YAML or JSON files for your resources and storing these definition files in a version control system like GitHub. This method is advantageous because it allows you to easily recreate resources by applying these files.

**Imperative Approach:** Sometimes resources are created directly using kubectl commands. To ensure these are backed up, you can query the Kubernetes API server to get the configurations of all resources.

1. **Persistent Storage:** If your applications use persistent storage, you should also back up the data stored in these volumes. This can be done using traditional backup tools or cloud-specific services.
2. **Etcd Database:** You can back up the etcd database by taking an snapshot

**Section 7: Security**

Let's dive into the security primitives in Kubernetes, covering everything you need to ensure your cluster is secure.

**1. Securing the Host** Before we delve into Kubernetes-specific security, it's crucial to secure the underlying hosts that form the cluster. Here are key steps:

* **Disable Root Access**: Prevent direct root access to minimize the risk of privilege escalation.
* **Use SSH Keys**: Disable password-based authentication and use SSH keys for secure access.
* **Secure the Physical Infrastructure**: Implement measures like firewalls, intrusion detection systems, and regular updates.

**2. Controlling Access to the kube-apiserver**

The **kube-apiserver** is the central component in Kubernetes. It manages all interactions within the cluster, making it a critical point of defense.

**Authentication: Who Can Access the Cluster?**

* **Static Files**: Store user IDs and passwords in static files.
* **Tokens**: Use tokens for authentication.
* **Certificates**: Employ client certificates for secure access.

**Authorization: What Can Users Do?**

Once authenticated, users' actions are controlled by authorization mechanisms:

* **Role-Based Access Control (RBAC)**: Assign roles to users and groups with specific permissions.

**3. Securing Communication:** Communication between various components in Kubernetes is secured using TLS encryption. This includes interactions between: Etcd, kube-controller-manager, kube-scheduler, Kubelet, kube-proxy

**Authentication**

Kubernetes doesn't manage user accounts natively. Instead, it relies on external sources such as files, certificates, or third-party identity services (like LDAP) to manage users.

**Types of Authentication Mechanisms**

1. **Static Password and Token Files:** You can create a CSV file containing user details and use it as an authentication source. Simple but not secure for production
2. **Certificates:** Certificates provide a secure way of authenticating users and services in Kubernetes. Kubernetes components, such as kube-apiserver, kubelet, etcd, and others, use certificates to secure communication. Secure method for authenticating users and services
3. **Third-Party Authentication Providers**: Kubernetes supports integration with external identity providers like LDAP, Kerberos, etc. This allows organizations to leverage existing authentication systems.

**Certificates in Kubernetes**

**Types of Certificates**

1. **Server Certificates**: Used by servers to establish secure communication with clients. Examples: apiserver.crt, etcdserver.crt, kubelet.crt.
2. **Client Certificates**: Used by clients to authenticate themselves to servers. Examples: admin.crt, scheduler.crt, kubeproxy.crt.
3. **Root Certificates** (CA Certificates): Used by the Certificate Authority (CA) to sign server and client certificates. Example: ca.crt.

**Naming Convention for Certificates**

* **Public Certificates**: Generally have .crt or .pem extensions. Examples: server.crt, client.crt, ca.crt.
* **Private Keys**: Generally have .key extensions or contain -key in the name. Examples: server.key, client-key.pem.

**Inter-Component Communication**

1. **kube-apiserver to etcd Server**: The kube-apiserver accesses the etcd server to manage cluster state. Can use the same certificates or generate new ones.
2. **kube-apiserver to kubelet**: Monitors and manages worker nodes. Can use the same certificates or generate new ones.

**Generating Certificates**

**Certificate Authority (CA)**

1. **Generate CA Certificate and Key**: Example: ca.crt and ca.key.
2. **Sign Certificates**: Use the CA to sign server and client certificates.

**Certification Details**

As a new administrator joining a Kubernetes team, you may be tasked with performing a health check on all the certificates in the cluster, especially if there are issues related to certificates.

**Step-by-Step Certificate Health Check**

1. **Identify Certificate Files:** Start by identifying all the certificate files used in the system.
2. **Extract Certificate Details:** For each certificate, extract the following details:

**Path**: Location of the certificate file.

**Names**: Common Name (CN) and any Subject Alternative Names (SANs).

**Organization**: The organization to which the certificate belongs.

**Issuer**: The CA that issued the certificate.

**Expiration Date**: The date when the certificate expires.

1. **Verify Certificate Details:** Ensure the following:

**Correct Names**: The names (CN and SANs) should match the expected values.

**Correct Organization**: Ensure the certificate belongs to the correct organization.

**Right Issuer**: Verify that the issuer is the expected CA.

**Not Expired**: Check that the certificate is still valid and not expired.

1. **Document Certificate Information:** Create a document (e.g., an Excel spreadsheet) with the following columns to track the certificate details: File Path, Common Name (CN), Alternate Names (SANs), Organization, Issuer, Expiration Date

**Kubeconfig**

A **kubeconfig** file is a configuration file used by kubectl and other Kubernetes tools to determine how to connect to the cluster. It contains details about clusters, users, and contexts. **Why Use kubeconfig?**

Instead of specifying server addresses, certificates, and other options in every kubectl command, you can consolidate this information into the kubeconfig file. This makes the process more efficient and less error-prone.

**Structure of a kubeconfig File:** A kubeconfig file has three main sections:

1. **Clusters**: Information about Kubernetes clusters.
2. **Users**: Credentials for accessing the clusters.
3. **Contexts**: Associations between clusters and users.

Each section is an array, allowing multiple entries for clusters, users, and contexts.

**Example kubeconfig File**

apiVersion: v1

kind: Config

clusters:

- cluster:

server: https://my-kube-playground:6443

certificate-authority: /path/to/ca.crt

name: my-kube-playground

users:

- name: my-kube-admin

user:

client-certificate: /path/to/admin.crt

client-key: /path/to/admin.key

contexts:

- context:

cluster: my-kube-playground

user: my-kube-admin

namespace: default

name: my-kube-admin@my-kube-playground

current-context: my-kube-admin@my-kube-playground

**Authorization**

Authorization in Kubernetes determines what actions a user or a service can perform within the cluster.

**Why Do You Need Authorization?**

While as an administrator you have full access to the cluster, you need to ensure that other users and applications have only the permissions they require. This is important for several reasons:

1. **Security**: Prevent unauthorized access and changes to the cluster.
2. **Separation of Duties**: Different teams (e.g., developers, testers) have different responsibilities and need different levels of access.
3. **Resource Management**: Ensure that users can only interact with resources within their namespace or assigned scope.

**Node Authorization:** Node authorization is a special authorizer that handles requests from kubelets. It ensures that kubelets can only access resources associated with their nodes, such as pods and nodes.

**Attribute-Based Access Control (ABAC)**

ABAC associates users or groups with a set of permissions defined in a policy file. Each policy is a JSON object specifying what actions a user or group can perform on which resources.

**Role-Based Access Control (RBAC)**

RBAC is a more flexible and manageable approach than ABAC. It involves defining roles and assigning them to users or groups.

1. **Role**: A set of permissions (verbs) on resources within a namespace.
2. **ClusterRole**: A set of permissions that apply cluster-wide.
3. **RoleBinding**: Assigns a Role to a user or group within a namespace.
4. **ClusterRoleBinding**: Assigns a ClusterRole to a user or group cluster-wide.

**Webhook Authorization**

Webhook authorization allows you to outsource authorization decisions to an external service. Kubernetes makes an API call to the external service with the user's request details, and the service decides whether to allow or deny the request.

**Service Accounts**

In Kubernetes, service accounts are essential for managing authentication for applications and services. **Types of Accounts**

1. **User Accounts**: Used by humans, such as administrators or developers, to perform tasks in the cluster.
2. **Service Accounts**: Used by applications, services, or automated tools to interact with the Kubernetes API.

**Creating and Using Service Accounts**

**Step 1: Create a Service Account**

**Step 2:** When a service account is created, a token is automatically generated and stored as a secret.

**Default Service Account:** Each namespace has a default service account automatically created. If no service account is specified for a pod, Kubernetes uses the default service account.

**Image Security**

**Where Are Images Pulled From?**

Since no registry is specified, Kubernetes pulls the image from Docker Hub, which has the default DNS name docker.io.

Other popular registries include:

* **Google Container Registry (GCR)**: gcr.io
* **Amazon Elastic Container Registry (ECR)**: account-id.dkr.ecr.region.amazonaws.com
* **Azure Container Registry (ACR)**: myregistry.azurecr.io

**Using Private Registries:** When you have applications that should not be publicly accessible, you can use private registries provided by cloud providers like AWS, Azure, or GCP. These registries can be made private, requiring credentials to access the images.

**Network Policies**

Network policies in Kubernetes are used to control the flow of traffic between pods, services, and other network endpoints in a cluster.

**Basic Concepts: Ingress and Egress Traffic**

To understand network policies, let's start with basic networking concepts:

**Example: Web Application and Database Server**

1. **Web Server**: Serves frontend to users.
2. **App Server**: Serves backend APIs.
3. **Database Server**: Stores data.

**Traffic Flow**

* **User to Web Server**: Ingress on port 80.
* **Web Server to App Server**: Egress from port 80 to ingress on port 5000.
* **App Server to Database Server**: Egress from port 5000 to ingress on port 3306.

**Rules for Traffic Flow**

1. **Web Server**:
   * Ingress Rule: Accept HTTP traffic on port 80.
   * Egress Rule: Allow traffic to port 5000 on the App Server.
2. **App Server**:
   * Ingress Rule: Accept traffic on port 5000.
   * Egress Rule: Allow traffic to port 3306 on the Database Server.
3. **Database Server**:
   * Ingress Rule: Accept traffic on port 3306.

**Implementing Network Policies:** Network policies in Kubernetes are used to define rules for ingress and egress traffic for pods. They allow you to control which traffic is allowed to and from your pods.

**Container Storage Interface (CSI)**

The Container Storage Interface (CSI) is a standard that allows container orchestration systems like Kubernetes to interact with different storage systems in a consistent way.

Initially, Kubernetes used Docker as the only container runtime, and all the code to interact with Docker was embedded within Kubernetes. As new container runtimes like RKT and CRI-O emerged, it became necessary to support these without modifying Kubernetes’ core code. This led to the creation of the Container Runtime Interface (CRI). Similarly, to support various networking solutions, the Container Networking Interface (CNI) was introduced. Following this pattern, the Container Storage Interface (CSI) was developed to support multiple storage solutions.

**How Does CSI Work?**

CSI defines a set of Remote Procedure Calls (RPCs) that the container orchestrator (like Kubernetes) uses to communicate with storage drivers. Here’s a simplified overview of how it works:

1. **Create Volume**: When a pod needs a volume, Kubernetes calls the CreateVolume RPC, passing details like the volume name. The storage driver handles this request, provisions a new volume, and returns the result.
2. **Delete Volume**: When a volume is no longer needed, Kubernetes calls the DeleteVolume RPC. The storage driver then decommissions the volume and returns the result.
3. **Attach/Detach Volume**: When a pod is scheduled on a node, Kubernetes calls the ControllerPublishVolume (attach) and ControllerUnpublishVolume (detach) RPCs to manage the volume’s attachment to the node.
4. **Mount/Unmount Volume**: Kubernetes calls the NodeStageVolume (mount) and NodeUnstageVolume (unmount) RPCs to manage the volume’s mounting on the node.

**Examples of CSI Drivers:** Many storage vendors have developed CSI drivers, including:Portworx**,** Amazon EBS**,** Azure Disk**,** Dell EMC (Isilon, PowerMax, Unity, XtremIO)**, x**NetApp, Nutanix, HPE, Hitachi, Pure Storage

**Volumes**

**Volumes in Kubernetes:** Kubernetes also uses volumes to persist data, similar to Docker:

* **Transient Pods**: Like Docker containers, Kubernetes pods are also temporary. When a pod is deleted, its data is lost unless it is stored in a volume.
* **Attaching Volumes**: To retain data, you attach a volume to a pod. Data generated by the pod is stored in the volume, so it remains even after the pod is deleted

**Volume Storage Options:** Kubernetes supports various storage options for volumes:

* **HostPath**: Uses a directory on the host. Suitable for single-node clusters but not recommended for multi-node clusters because each node has its own separate directory.
* **Network File Systems**: Options like NFS, CephFS, and GlusterFS allow shared storage across multiple nodes.
* **Cloud Storage**: Public cloud solutions like AWS EBS, Azure Disk, and Google Persistent Disk provide scalable and reliable storage.

Persistent volumes and Persistent Volume Claim

**What are Persistent Volumes (PVs)?**

Persistent Volumes (PVs) are a way to manage storage in a Kubernetes cluster. Unlike regular volumes that are defined within a pod’s specification, PVs are cluster-wide resources that are created and managed by administrators.

In large environments with many users and pods, managing storage individually for each pod can be difficult. Persistent Volumes allow administrators to create a pool of storage that users can request as needed.

**Creating a Persistent Volume Claim (PVC):** A Persistent Volume Claim (PVC) is a request for storage by a user. It allows users to request specific storage resources without needing to know the details of the underlying storage.

**Binding PVs and PVCs:** When a PVC is created, Kubernetes looks for a matching PV based on the requested storage and access modes. If a suitable PV is found, it binds the PVC to the PV. If no suitable PV is available, the PVC remains in a pending state until a matching PV is created.

**Storage Classes:** Storage Classes provide a way to define different types of storage (e.g., SSDs, HDDs) and their properties

**Storage Classes**

**Static Provisioning:** In static provisioning, you manually create a Persistent Volume (PV) and then create a Persistent Volume Claim (PVC) to use that PV.

**Dynamic Provisioning:** Dynamic provisioning automates the creation of PVs when a PVC is made. This is where Storage Classes come in.

**What are Storage Classes?**

A **Storage Class** in Kubernetes defines a way to dynamically provision storage. It specifies a provisioner and parameters for the type of storage you want to create.

**Using a Storage Class with PVCs**

When you create a PVC, you can specify the storage class to use. This tells Kubernetes to use the specified storage class to dynamically provision the storage.

**How Dynamic Provisioning Works**

1. **Create PVC**: When a PVC is created with a specified storage class, Kubernetes uses the storage class’s provisioner to create a new disk on the cloud provider.
2. **Provision PV**: The provisioner automatically creates a PV that references the newly created disk.
3. **Bind PVC to PV**: Kubernetes binds the PVC to the newly created PV.

**Example Provisioners**

* **Google Cloud**: kubernetes.io/gce-pd
* **AWS EBS**: kubernetes.io/aws-ebs
* **Azure Disk**: kubernetes.io/azure-disk
* **CephFS**: ceph.com/cephfs
* **Portworx**: pxd.portworx.com
* **ScaleIO**: scaleio

**Section 9: Networking**

**Required Ports**

**Required Ports:** Certain ports need to be open for the various components of the Kubernetes control plane to communicate. Here are the key ports:

1. **API Server (Master Node)**:
   * **Port 6443**: This is the main port for the Kubernetes API server. It is used by kubelets on worker nodes, kubectl (command-line tool), external users, and other control plane components to communicate with the API server.
2. **Kubelet (Master and Worker Nodes)**:
   * **Port 10250**: Kubelets, which are agents running on each node, listen on this port. Kubelets can be present on both master and worker nodes.
3. **Kube Scheduler (Master Node)**:
   * **Port 10259**: The kube scheduler, which assigns pods to nodes, requires this port to be open.
4. **Kube Controller Manager (Master Node)**:
   * **Port 10257**: The kube controller manager, which runs various controllers to manage the state of the cluster, requires this port to be open.
5. **Worker Nodes (External Access)**:
   * **Ports 30000-32767**: These ports are used to expose services running on the worker nodes to external users.
6. **ETCD Server (Master Node)**:
   * **Port 2379**: ETCD is the key-value store used by Kubernetes to store all cluster data. This port is used for communication with the ETCD server.
   * **Port 2380**: If you have multiple master nodes, this port is used for communication between ETCD instances.

**Setting Up Networking:** When setting up networking for your Kubernetes nodes, you need to ensure that these ports are open in your firewalls, IP table rules, or network security groups, especially if you are using a cloud environment like Google Cloud Platform (GCP), Azure, or AWS.

**Pod Networking**

When you set up a Kubernetes cluster, you need to address not only the network connectivity between the nodes but also the networking at the pod level. This involves ensuring that pods can communicate with each other within the cluster and that services running on these pods can be accessed both internally and externally.

**Implementing Pod Networking:** To meet these requirements, you need to implement a networking solution. Let’s go through the steps to set up a basic networking model using bridge networks and routing.

**Step 1: Network Interfaces and IP Addresses:** Each node in your cluster has a network interface with an IP address. For example:Node 1: 192.168.1.11

**Step 2: Create Bridge Networks:** On each node, create a bridge network to connect the pods:Node 1: 10.240.1.0/24

**Step 3: Attach Containers to the Bridge Network:** For each container, create a virtual network interface and attach it to the bridge network. This can be automated using a script.

**Step 4: Enable Cross-Node Communication:** To enable pods on different nodes to communicate, add routes on each node to direct traffic to the appropriate bridge network.

**Step 5: Automate with CNI (Container Network Interface):** To automate the network setup for each pod, use CNI plugins. CNI provides a standard for network configuration and management.

**CNI and Weave**

CNI defines how container runtimes (like Kubernetes) should set up networking for containers.

**How Kubernetes Uses CNI Plugins**

1. **Container Runtime**: Kubernetes uses container runtimes like Containerd or CRI-O to manage containers.
2. **Network Plugins**: These runtimes use CNI plugins to set up networking for the containers.

**Weave CNI Plugin**

**What is Weave?**

Weave is a CNI plugin that provides a simple way to connect containers across multiple hosts. It creates a virtual network that spans all the nodes in your Kubernetes cluster.

**How Weave Works**

1. **Weave Agents**: When you deploy Weave, it installs agents (or peers) on each node in the cluster.
2. **Communication**: These agents communicate with each other to share information about the network topology and the pods running on each node.
3. **Bridge Network**: Weave creates a bridge network on each node, typically named weave.
4. **IP Assignment**: Weave assigns IP addresses to pods from a predefined range.

**Packet Flow with Weave**

1. **Sending Packets**: When a pod sends data to another pod on a different node, the Weave agent intercepts it.
2. **Encapsulation**: The agent wraps the data in a new packet and sends it to the target node.
3. **Receiving Packets**: The agent on the target node unwraps the packet and delivers it to the correct pod.

**IP Address Management**

When we talk about IP address management in Kubernetes, we’re focusing on how IP addresses are assigned to pods, not the nodes themselves. Here’s how it works:

**Who Manages IP Addresses?**

The responsibility for assigning IP addresses to pods falls on the CNI (Container Network Interface) plugins. These plugins are part of the network solution you choose for your Kubernetes cluster.

**How IP Addresses are Assigned**

1. **CNI Plugins**: When a pod is created, the CNI plugin assigns an IP address to it. This process is defined by the CNI standards.
2. **IPAM (IP Address Management)**: The CNI configuration file includes an IPAM section that specifies how IP addresses should be managed.

**DNS in Kubernetes**

**What is DNS?**

DNS (Domain Name System) translates human-readable domain names (like example.com) into IP addresses that computers use to identify each other on the network. In Kubernetes, DNS is used to resolve the names of services and pods to their respective IP addresses.

**DNS in Kubernetes**

Kubernetes includes a built-in DNS server that helps pods and services within the cluster resolve each other’s names to IP addresses. This DNS server is typically implemented using CoreDNS.

**CoreDNS**: Kubernetes uses CoreDNS as the DNS server. CoreDNS is deployed as a pod within the cluster and handles all DNS queries.

**DNS Resolution in Kubernetes**

**Nodes:** Each node in your Kubernetes cluster has a name and an IP address. These are usually registered in your organization’s DNS server

**How DNS Works in Kubernetes**

1. **Service DNS Records**: When a service is created, Kubernetes automatically creates a DNS record for it. This record maps the service name to its IP address. For example, the web-service can be accessed using its name web-service.
2. **Namespace and DNS**: Kubernetes uses namespaces to organize resources. If a service is in a different namespace, you need to use the full name to access it. For example, if web-service is in the apps namespace, you would access it using web-service.apps.
3. **Fully Qualified Domain Name (FQDN)**: Each service can be accessed using its fully qualified domain name (FQDN). The FQDN includes the service name, namespace, and the cluster domain. For example, web-service.apps.svc.cluster.local

**Pod DNS Records:** By default, Kubernetes does not create DNS records for pods. However, you can enable this feature. When enabled, Kubernetes generates DNS names for pods by converting their IP addresses to a specific format. For example, a pod with IP 10.244.1.5 would have a DNS name 10-244-1-5.default.pod.cluster.local

**Ingress**

**Ingress** in Kubernetes is a resource that manages external access to services within a cluster. It allows you to expose multiple services using a single IP address

**Why Use Ingress?**

Without Ingress, you would need to expose each service using a separate **NodePort** or **LoadBalancer** service, which can be inefficient and difficult to manage.

**Example Scenario:** Imagine you have an online store and a video streaming service running in your Kubernetes cluster.

1. **Deploying the Applications**:
   * **Online Store**: Deployed as a pod and exposed using a service.
   * **Video Streaming**: Deployed as a separate pod and exposed using another service.
   * Just like that if we add some new feature we would need another service ti expose
2. **Using Ingress**:
   * **Single URL**: Instead of exposing each service separately, you use Ingress to expose both services using a single URL like myonlinestore.com

**How Ingress Works**

1. **Ingress Resource**: You define an Ingress resource with rules that specify how to route traffic. For example, you might route traffic from example.com/app1 to one service and traffic from example.com/app2 to another service.
2. **Ingress Controller**: The Ingress controller watches for changes to Ingress resources and updates the routing configuration accordingly. It ensures that incoming requests are routed to the correct service based on the rules defined. **Choose an Ingress Controller**: There are several Ingress controllers available, such as NGINX, Traefik, HAProxy, and Istio

**Example of Ingress Rules**

1. **Domain-Based Routing**:
   * **Rule 1**: Route traffic from myonlinestore.com to the online store application.
   * **Rule 2**: Route traffic from watch.myonlinestore.com to the video streaming application.
2. **Path-Based Routing**:
   * **Rule 1**: Route traffic from myonlinestore.com/app1 to the online store application.
   * **Rule 2**: Route traffic from myonlinestore.com/app2 to the video streaming application.

**Kubeadm**

**What is kubeadm?**

**kubeadm** also known as Kubernetes Admin is a tool that helps you set up a Kubernetes cluster by automating the installation and configuration of the various components needed for a Kubernetes cluster. It simplifies the process of bootstrapping a Kubernetes cluster by handling the complex tasks of setting up certificates, configuring networking, and installing necessary components.

**Steps to Set Up a Kubernetes Cluster with kubeadm**

* 1. **Provision Multiple Systems or VMs:** First, you need to have multiple systems or virtual machines (VMs) ready. These can be physical machines or VMs running on your laptop or in the cloud. You will need at least two nodes: one master node and one or more worker nodes.
  2. **Designate Master and Worker Nodes:** Decide which node will be the master and which ones will be the worker nodes. The master node will run the control plane components, while the worker nodes will run your application workloads.
  3. **Install a Container Runtime:** Install a container runtime on all nodes. For example: **containerd**.
  4. **Install kubeadm, kubelet, and kubectl:** Install the kubeadm, kubelet, and kubectl tools on all nodes.
  5. **Initialize the Master Node:** On the master node, initialize the Kubernetes cluster using kubeadm init. This command sets up the control plane components (API server, etcd, controller manager, scheduler) on the master node.
  6. **Set Up the Pod Network:** Before adding worker nodes, you need to set up the pod network. Kubernetes requires a special networking solution to allow pods to communicate across nodes. For this example, we will use Flannel.
  7. **Join Worker Nodes to the Cluster:** On each worker node, run the kubeadm join command provided by the kubeadm init output on the master node. This command connects the worker nodes to the master node
  8. **Verify the Cluster:** Once all nodes have joined the cluster, you can verify the setup by running the following command on the master node: kubectl get nodes. You should see the master and worker nodes listed and ready**.**