**Kubernetes**

**What is difference between docker and kubernetes?**

**Docker:**

**What is it?** Docker is like a lunchbox for your applications. It's a technology that allows you to package your application and all its dependencies into a single, portable container.

**Isolation:** Docker provides a way to isolate your application from the underlying system. It ensures that your app runs consistently across different environments.

**Runs a Single Container:** Docker is primarily used to run and manage individual containers. A container is a lightweight, standalone, and executable package that includes everything needed to run your application.

**Easier for Developers:** It's great for developers because they can create, test, and deploy applications in the same environment, which reduces the "it works on my machine" problem.

**Kubernetes:**

**What is it?** Kubernetes is like an orchestra conductor for containers. It's an open-source system for automating the deployment, scaling, and management of containerized applications.

**Orchestration:** Kubernetes manages the deployment and scaling of containers in a more complex and automated way. It can balance the load, handle failures, and ensure high availability.

**Manages Multiple Containers:** Kubernetes can manage multiple containers, not just one. It's perfect for applications that are composed of many microservices running in multiple containers.

**Operations and Scaling:** It's mainly used by operations teams to manage large-scale containerized applications. Kubernetes can automatically scale up or down based on the demand, making it highly scalable and resilient.

In summary, Docker is like a container that holds your application and its dependencies, while Kubernetes is like a manager that handles multiple containers, making sure they run, scale, and recover efficiently. Docker is more for packaging and running individual applications, while Kubernetes is for orchestrating and managing many containers in complex, distributed applications. They can also be used together, with Kubernetes orchestrating Docker containers.

**Kubernetes Architecture:**

**Kubernetes Cluster:**

Imagine a Kubernetes cluster as a group of computers (usually servers or virtual machines) that work together to manage and run your containerized applications.

**Control Plane:**

The control plane is like the brain of the Kubernetes cluster. It's where all the decisions are made.

The master node in a Kubernetes architecture is used to manage the states of a cluster. It is actually an entry point for all types of administrative tasks. In the Kubernetes cluster, more than one master node is present for checking the fault tolerance.

It consists of four main components:

**API Server:**

Think of it as the command center. It receives instructions and commands from users and administrators.

**Scheduler:**

This component is like a matchmaker. It decides where to run your containers based on factors like available resources and constraints.

**Controller Manager:**

Imagine this as an automated overseer. It watches over your cluster and ensures that the desired state (what you want to run) matches the actual state (what is running).

**Node Controller**: Responsible for noticing and responding when nodes go down

**Replication Controller:** Responsible for maintaining the correct number of pods for every replication controller object in the system

**EndPoints Contoller:** Populates the Endpoints object (that is, its joins services and pods)

**Service Account and Token Controller:** Create defaults accounts and api access tokens for new namespaces

**ETCD**

It is an open-source, simple, distributed key-value storage which is used to store the cluster data. It is a part of a master node which is written in a GO programming language.

**Nodes (Worker Machines):**

Nodes are like the workers in the Kubernetes cluster. They do the actual work of running containers. Each node has two key components:

**Kubelet:** Think of this as the node's manager. It communicates with the control plane and ensures the containers are running on the node as expected.

**Container Runtime:** This is like the worker's toolbox. It's software that actually runs the containers, such as Docker or container.

**Pods:**  A pod is a combination of one or more containers which logically execute together on nodes. One worker node can easily execute multiple pods.

**Kube-proxy:** It is the process responsible for forwarding the request from services to the pods. It has intelligent logic to forward the request to the right pod in the worker node

**Services**

Services are like the front doors to your applications. They provide a stable network endpoint for accessing your application, even if the containers inside the pods change or scale up or down.

**Ingress:**

Ingress acts as a traffic cop for incoming requests. It manages external access to services, such as routing requests to the correct service based on the URL.

**Config Maps and Secrets:**

These are like secret recipes. Config Maps hold configuration data, while Secrets hold sensitive data like passwords and keys. Applications can use these for their configuration needs.

In simple terms, Kubernetes is a system for managing and orchestrating containers (like Docker containers). It uses a control plane to make decisions about where and how containers should run and nodes (worker machines) to actually run them. Pods are the basic building blocks for your applications, and services and ingress help manage how your applications are accessed from the outside world. It's all about making sure your containerized applications run reliably, scale when needed, and can be managed easily.

**Let's illustrate the Kubernetes architecture with an everyday example:**

**Imagine a Restaurant:**

**Control Plane:** The control plane is like the restaurant manager. The manager takes orders from customers (API Server), decides where to seat them (Scheduler), and ensures that everything runs smoothly, such as assigning waitstaff to tables (Controller Manager).

**Nodes (Worker Machines):** Think of nodes as the tables or workstations where the actual cooking and serving happen. Each table (node) has a dedicated waiter (Kubelet) who communicates with the manager (control plane) and ensures that the dishes (containers) are prepared and served correctly. The kitchen (container runtime) is where the chef (Docker) prepares the food (containers).

**Pods**: Consider a pod as a tray or a plate. You don't serve individual items one by one; you put a group of items on a tray (pod). For example, a tray can hold a burger (container) and a side of fries (another container). These two items share the same tray (pod) and are served together.

**Services:** Services are like the menu or the order window in a restaurant. When you order a specific dish (service), it doesn't matter who is cooking it or where it's coming from (which pod or kitchen). The order window (service) ensures that your order is delivered to you, even if the chef (container) changes or is replaced.

**Ingress:** Ingress is similar to the restaurant's entrance and reservation system. It manages how customers (incoming requests) are directed to their reserved tables (services). For example, if you want a table for two (accessing a web application), the host (Ingress) guides you to your reserved spot (service).

**Config Maps and Secrets**: Think of Config Maps as the restaurant's menu, where you list your food preferences (configuration data). Secrets, on the other hand, are like the secret recipes that only the chef knows (sensitive data like passwords or encryption keys). These help customize your order (application) without revealing the restaurant's secrets.

In this restaurant analogy, Kubernetes is like an advanced restaurant management system that automates and streamlines the entire dining experience, from taking orders to serving food, all while ensuring the best customer experience. Just as the restaurant manager doesn't need to know how to cook each dish, Kubernetes abstracts away the complexities of managing containers and allows you to focus on your applications.

**Commands for Kubectl**

Here are some common commands for interacting with a Kubernetes cluster

**To view a list of all the pods in the cluster, you can use the following command:**

kubectl get pods

**To view a list of all the nodes in the cluster, you can use the following command:**

kubectl get nodes

**To view a list of all the services in the cluster, you can use the following command:**

kubectl get services

**Kubernetes Manifest**

A Kubernetes manifest file is **a YAML or JSON file that describes the desired state of a Kubernetes object**. These objects can include: Deployment, Replica set, Service

Manifest files define the specifications of the object, such as its: Metadata, Properties, Desired state.

Manifest files are a common name for any Kubernetes configuration file that defines the configuration of various Kubernetes components.

**Deployments:**

**Definition and Management of Desired State:** Deployments are a Kubernetes resource that allows you to define and manage your application's desired state. You specify how many instances (replicas) of your application should be running, which container image to use, and other configuration details.

**Scaling:** Deployments make it easy to scale your application up or down by adjusting the replica count. For example, if you initially specify three replicas, you can easily scale to five replicas to handle increased traffic.

**Rolling Updates:** Deployments support rolling updates, ensuring that when you need to change your application (e.g., update to a new version), Kubernetes replaces the old Pods with new ones gradually, ensuring continuous availability. This process is known as a rolling update, and it minimizes downtime.

**Rollback Strategies**: In case a new version of your application has issues, Deployments allow you to perform a rollback to a previous stable version. This helps maintain the health of your application.

**Services:**

**Pods Communication:** In Kubernetes, Pods can come and go due to scaling, updates, or failures. Services provide a stable network endpoint for a set of Pods, allowing other components to reliably communicate with them.

**Types of Services:** Kubernetes offers different types of Services, including ClusterIP (for internal cluster communication), NodePort (for exposing services on a specific node's IP address), LoadBalancer (for external load balancing if supported by your cloud provider), and ExternalName (for mapping a service to an external DNS name).

**Selectors:** Services use label selectors to determine which Pods they should route traffic to. By matching labels in the Service definition with labels on Pods, you can control how the Service directs traffic.

**Load Balancing:** When you have multiple Pods behind a Service, it automatically load balances traffic among them, ensuring that your application is highly available and scalable.

In summary, Deployments help you manage your application's lifecycle, from defining its desired state to rolling updates and rollbacks. Services, on the other hand, provide a reliable way to access your application by managing network endpoints for your Pods, ensuring that your application remains accessible and scalable even as Pods come and go.

**Deployment Manifest (deployment.yaml):**

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app-deployment

spec:

replicas: 3

selector:

matchLabels:

app: my-app

template:

metadata:

labels:

app: my-app

spec:

containers:

- name: my-app-container

image: your-container-image:tag

ports:

- containerPort: 80

**In this Deployment manifest:**

apiVersionspecifies the API version for Deployment.

kind specifies the resource type, which is a Deployment.

metadata includes the name of the Deployment.

spec defines the desired state of the Deployment, including the number of replicas (3 in this case) and the template for the Pods.

selector is used to match the labels with the Pods managed by the Deployment.

template defines the Pod template, including labels and the container with its image and port configuration.

**Service Manifest (service.yaml):**

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

selector:

app: my-app

ports:

- protocol: TCP

port: 80

targetPort: 80

type: LoadBalancer

**In this Service manifest:**

apiVersion specifies the API version for Service.

kind specifies the resource type, which is a Service.

metadata includes the name of the Service.

spec defines the desired state of the Service.

selector matches the labels with the Pods for which this Service will route traffic (in this case, Pods with the label app: my-app).

ports specify the port configuration. In this example, it exposes port 80 of the Service, which maps to port 80 of the Pods.

type indicates the type of the Service. In this case, it's set to LoadBalancer, which means it will be exposed externally if your Kubernetes cluster supports LoadBalancer services.

**You can create these resources in your Kubernetes cluster using the kubectl apply command**:

kubectl apply -f deployment.yaml

kubectl apply -f service.yaml

**ConfigMaps and Secrets:**

ConfigMaps Example: Let's say you have a web application that needs to know the backend API's URL. You can create a ConfigMap with a key-value pair like api-url: http://backend-service:8080, and then you can mount this as an environment variable in your web application's Pod.

Secrets Example: If your web application requires a database password, you can create a Secret called db-credentials with the key-value pair password: mysecretpassword, which can be used by your application securely.

**StatefulSets:**

Imagine you have a stateful application like a database. Using a StatefulSet, you can ensure that each instance (e.g., database server) is assigned a unique identifier (e.g., db-0, db-1) and they are started in order. This is crucial for databases where each instance has its data that shouldn't be overwritten.

**Volumes and Persistent Storage:**

Let's say you have a blogging platform where user-uploaded images need to be stored. You can use a Persistent Volume to ensure that these images are not lost even if the underlying Pod crashes or is rescheduled to another node.

**Namespaces:**

Consider a Kubernetes cluster shared by multiple teams or departments. You can create namespaces like marketing, development, and sales to separate resources. This way, each team can work independently within their designated namespaces.

**Labels and Selectors:**

Suppose you have multiple versions of your application running in a cluster. You can label the Pods with their version (e.g., app-version: v1, app-version: v2) and then use selectors to filter and manage specific versions. For example, you could scale up only the Pods labeled with app-version: v2.

**Resource Management:**

Let's say you have a web application with multiple containers in a Pod. By setting resource requests and limits, you can ensure that one container doesn't consume all the available resources, potentially affecting other containers. For instance, you can specify that the frontend container should request at least 0.5 CPU and 256MB of memory and limit it to 1 CPU and 512MB of memory.

These practical examples should help you understand how ConfigMaps, Secrets, StatefulSets, Persistent Storage, Namespaces, Labels, Selectors, and Resource Management are used in real-world scenarios to manage configuration, stateful applications, storage, resource allocation, resource organization, and security in a Kubernetes cluster.

**Custom Resource Definitions (CRDs):**

Example: Imagine you're managing a large e-commerce platform on Kubernetes, and you need to handle "Product" resources with specific attributes like name, price, and availability. You can create a CRD named "Product" to define these custom resources and enable Kubernetes to understand and manage them.

**Operators:**

Example: Let's say you have a complex distributed application like a Cassandra database. You can create a custom Cassandra Operator that automates tasks like scaling, backups, and failover. The Operator watches for changes in the custom resource "CassandraCluster" and takes actions accordingly.

**Network Policies:**

Example: In a microservices application, you want to restrict communication between different components. You can create a Network Policy that allows only the "Frontend" Pods to communicate with the "Backend" Pods and blocks all other traffic, enhancing security and isolation.

**Security:**

Example: You can use PodSecurityPolicies to define security policies, such as disallowing privileged containers or requiring specific SELinux labels. RBAC can be used to control who can access and modify resources. For instance, you can grant only specific users or service accounts the ability to scale deployments or access secrets.

**Helm:**

Example: If you have a complex application with multiple microservices, databases, and configurations, you can create a Helm chart for it. This chart can include all the resources and configurations needed to deploy the entire application. Users can then deploy the application with a single Helm command, simplifying the deployment process.

**Monitoring and Logging:**

Example: To monitor your Kubernetes cluster and applications, you can use Prometheus for collecting metrics, Grafana for visualization, and Alertmanager for alerting. For logging, you can set up the EFK (Elasticsearch, Fluentd, Kibana) stack to collect and analyze logs generated by your containers.

**Advanced Networking:**

Example: An Ingress controller allows you to expose multiple services using a single external IP address and hostname. For instance, you can configure an Ingress to route traffic to different services based on the path in the URL. Service Mesh, like Istio, can be used to control traffic, manage encryption, and observe communication between microservices in a complex application.

**Multi-Cluster and Federation:**

Example: In a global deployment scenario, you might have clusters in different regions. You can use kubefed to manage these clusters and deploy applications across multiple regions. This enables you to provide low-latency access to users in different parts of the world.

**Custom Admission Controllers:**

Example: Suppose you want to enforce a policy where every Pod must have a label indicating its environment (e.g., "production" or "development"). You can create a custom admission controller that checks incoming Pod requests and adds the necessary label if it's missing or rejects the request if it doesn't comply with the policy.

These examples should help you understand how these advanced Kubernetes concepts are applied in real-world scenarios to extend, automate, secure, manage, and scale your Kubernetes infrastructure and applications.

**Security:**

Security in Kubernetes is all about protecting your applications and data from unauthorized access, vulnerabilities, and malicious activities. Here are some key aspects of security in Kubernetes:

**Authentication:** This is about verifying the identity of users and components in your cluster. Kubernetes uses various authentication methods, such as client certificates, tokens, and service accounts, to ensure that only authorized entities can access the cluster.

**Authorization:** Once a user or component is authenticated, Kubernetes enforces authorization policies to determine what actions they can perform. Role-Based Access Control (RBAC) is commonly used to define these policies, ensuring that users and components have the right permissions.

**Network Policies:** Network policies allow you to control the communication between Pods. By specifying rules, you can restrict which Pods can talk to each other, enhancing security and isolation.

**Pod Security:** PodSecurityPolicies are used to define security policies for Pods. You can enforce restrictions like not allowing privileged containers, specifying SELinux options, or limiting host resources.

**Secrets Management:** Kubernetes offers a way to store sensitive information, such as passwords and tokens, in secrets. These secrets are base64 encoded and can be mounted as environment variables or volumes in Pods.

**Security Context:** You can define security contexts for Pods, specifying attributes like user, group, and SELinux options to enhance security.

**Container Scanning and Vulnerability Management:** It's important to regularly scan container images for vulnerabilities and apply patches to ensure your applications are secure.

**Cluster Updates:** Keeping your Kubernetes cluster up to date with security patches is crucial to protect against known vulnerabilities.

**Helm:**

Helm is like a Swiss Army knife for managing applications on Kubernetes. Here's how it works:

**Packaging:** Helm allows you to package your entire application, including Kubernetes manifests, configurations, and dependencies, into a single unit called a "chart." This chart is like a recipe for your application

**Templating:** Helm uses Go templates to create dynamic Kubernetes manifests based on the values you provide. This enables you to customize your deployment for different environments or configurations.

**Deployment:** You can deploy charts using Helm with a single command. Helm takes care of creating all the required Kubernetes resources and configurations for your application.

**Version Control:** Helm allows you to version your charts, making it easy to track changes and roll back to previous versions if needed.

**Sharing and Community:** Helm has a vast library of pre-built charts that you can use to quickly deploy common applications. You can also share your own charts with the Kubernetes community.

In a nutshell, Helm streamlines the deployment and management of applications on Kubernetes by packaging them into charts and providing a consistent way to deploy and configure them. It's like a convenient package manager for your Kubernetes applications.

**1.What is the architecture of kuberetes**

The architecture of Kubernetes is based on a master-worker model, consisting of a control plane (master) and multiple nodes (workers). It includes components like the API server, etcd, kube-scheduler, kube-controller-manager, and the kubelet on nodes.

**2.What does control manager,etch, scheduler, API server do**

Control Manager components include the kube-controller-manager, which manages various controller processes. Etcd is a distributed key-value store. The scheduler schedules work to nodes. The API server serves the Kubernetes API.

**3.﻿﻿﻿What is a manifest file and what are the components of it**

A manifest file in Kubernetes defines the desired state of a resource. It typically includes metadata, specifications, and labels. Components of a manifest may include pods, services, deployments, and more.

**4.What is node affinity, pod afinity, taint toleration**

Node affinity specifies conditions for pod placement. Pod affinity and anti-affinity control pod scheduling based on affinity rules. Taints and tolerations are used to control which pods can be scheduled on a node based on node taints.

**5.﻿﻿﻿What is node port, cluster ip**

NodePort is a way to expose a service on a specific port on each node. ClusterIP exposes a service on a cluster-internal IP.

**6.﻿﻿﻿What is persitant volumes and why we use it**

Persistent Volumes (PVs) in Kubernetes are used to store data that persists beyond the lifecycle of a pod. They provide a way to decouple storage from pods and ensure data persistence.

**7.﻿﻿﻿Describe what is pod and what is pod lifecycle**

A pod in Kubernetes is the smallest deployable unit, typically containing one or more containers. The pod's lifecycle includes phases like Pending, Running, Succeeded, or Failed.

**8.﻿﻿﻿What are the components on master and worked node**

Components on the master node include the API server, etcd, kube-scheduler, and kube-controller-manager. Worker nodes have the kubelet, kube-proxy, and container runtime.

**9.﻿﻿﻿What is ingress controller**

An Ingress Controller manages external access to services within a cluster. It typically routes incoming traffic to the appropriate services.

**10.﻿﻿﻿﻿What are types of services in kuberntes**

Kubernetes offers several types of services, including ClusterIP, NodePort, LoadBalancer, and ExternalName, to expose applications in different ways.

**11.How one pod talks with other pod**

Pods communicate with each other within a cluster using services or directly via IP addresses and ports.

**12.﻿﻿﻿﻿How the pod healthcheck is done(describe rediness, livesness)**

Pod health checks are performed using readiness and liveness probes. Readiness checks if a pod is ready to serve traffic, while liveness checks if it's running correctly.

**13.How the monitoring is done(integration on Prometheus and grafana)**

Monitoring in Kubernetes can be done by integrating with tools like Prometheus and Grafana to collect and visualize metrics from the cluster.

**14.﻿﻿﻿﻿What is demonset, replicaset, horizontal pod autoscaler**

DaemonSet ensures that a copy of a pod runs on each node, ReplicaSet maintains a specified number of pod replicas, and Horizontal Pod Autoscaler automatically scales the number of pods based on CPU utilization.

**15.Write a manifest file of your own choice**

A manifest file example for a simple Nginx deployment:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

**16.﻿﻿﻿﻿What is namespace and why we use it**

A Namespace in Kubernetes is a way to organize and isolate resources within a cluster. It helps manage and segregate workloads for various teams or projects.

**17. ﻿﻿﻿﻿What are helm charts and uses**

Helm charts are packages used to define, install, and upgrade even the most complex Kubernetes applications. They simplify application deployment and management.