**1. You are tasked with creating a simple pod in your Kubernetes cluster. The pod should run a container using the busybox image.**



Create a YAML Manifest:

Create a YAML manifest file **(Firstpod.yaml)** with the following content:

***apiVersion: v1***

***kind: Pod***

***metadata:***

***name: myapp***

***labels:***

***name: myapp***

***spec:***

***containers:***

***- name: myapp***

***image: busybox***

***args:***

***- "/bin/sh"***

***- "-c"***

***- "sleep 3600"***

***resources:***

***limits:***

***memory: "128Mi"***

***cpu: "500m"***

***ports:***

***- containerPort: 8080***

***Apply the Manifest:***

***Apply the YAML manifest using the command:***

***K apply -f .\Firstpod.yaml***



Verify the Pod:

Verify that the Pod has been created successfully by running:

**k get pods**



**2. Change the image name from busybox to nginx, also check that pod is running well.**

Edit the YAML Manifest:

Edit the YAML manifest file **(Firstpod.yaml)** to change the image name from busybox to nginx. Here's the modified manifest:

***apiVersion: v1***

***kind: Pod***

***metadata:***

***name: myapp***

***labels:***

***name: myapp***

***spec:***

***containers:***

***- name: myapp***

***image: nginx***

***args:***

***- "/bin/sh"***

***- "-c"***

***- "sleep 3600"***

***resources:***

***limits:***

***memory: "128Mi"***

***cpu: "500m"***

***ports:***

***- containerPort: 8080***

Apply the Manifest Changes:

Apply the updated YAML manifest to the Kubernetes cluster using the kubectl apply command:

**k apply -f .\Firstpod.yaml**



Verify the Pod Status:

Check the status of the Pod to ensure it's running and the new container using the nginx image is created:

**k get pods**



**3. Create a ReplicaSet named " app-replicaset " managing three replicas of an application pod using the nginx image.**

Create a YAML Manifest:

Create a YAML manifest file (nginxreplica.yaml) with the following content:

***apiVersion: apps/v1***

***kind: ReplicaSet***

***metadata:***

***name: app-replicaset***

***labels:***

***app: app-replicaset***

***spec:***

***replicas: 3***

***selector:***

***matchLabels:***

***app: app-replicaset***

***template:***

***metadata:***

***labels:***

***app: app-replicaset***

***spec:***

***containers:***

***- name: app-replicaset***

***image: nginx***

***ports:***

***- name: web***

***containerPort: 80***

***protocol: TCP***

***env:***

***- name: NGX\_VERSION***

***value: 1.16.1***

Apply the Manifest:

Apply the YAML manifest to the Kubernetes cluster using the command:

**k apply -f .\nginxreplica.yaml –validate=false**



Verify the ReplicaSet:

Check the status of the ReplicaSet to ensure it's created successfully:

**k get replicasets**



**4. Create a Deployment named "app-deployment" managing four replicas of an application pod using the nginx:alpine image.**

Create a YAML Manifest:

Create a YAML manifest file (nginx-alpine.yaml) with the following content:

***apiVersion: apps/v1***

***kind: Deployment***

***metadata:***

***name: app-deployment***

***namespace: default***

***labels:***

***app: app-deployment***

***spec:***

***selector:***

***matchLabels:***

***app: app-deployment***

***replicas: 4***

***strategy:***

***rollingUpdate:***

***maxSurge: 25%***

***maxUnavailable: 25%***

***type: RollingUpdate***

***template:***

***metadata:***

***labels:***

***app: app-deployment***

***spec:***

***# initContainers:***

***# Init containers are exactly like regular containers, except:***

***# - Init containers always run to completion.***

***# - Each init container must complete successfully before the next one starts.***

***containers:***

***- name: app-deployment***

***image: nginx:alpine***

***resources:***

***requests:***

***cpu: 100m***

***memory: 100Mi***

***limits:***

***cpu: 100m***

***memory: 100Mi***

***livenessProbe:***

***tcpSocket:***

***port: 80***

***env:***

***- name: DB\_HOST***

***valueFrom:***

***configMapKeyRef:***

***name: app-deployment***

***key: DB\_HOST***

***ports:***

***- containerPort: 80***

***name: app-deployment***

Apply the Manifest:

Apply the YAML manifest to the Kubernetes cluster using the command:

**k apply -f .\nginx-alpine.yaml**



Verify the Deployment:

Check the status of the Deployment to ensure it's created successfully:

**k get deployments**



**5. Explain how to automatically roll back to the previous version using the "app-deployment."**

Automatically rolling back to the previous version using a Kubernetes Deployment can be achieved by taking advantage of the Deployment's rollout feature. Kubernetes Deployment allows you to manage updates to your application by controlling the rollout process.

Here's how you can configure automatic rollback to the previous version using a Deployment:

**Define the Deployment:** First, define your Deployment manifest (app-deployment.yaml), including the desired image version, replicas, and other specifications.

**Update the Deployment:** When you need to update the application to a new version, apply the new Deployment manifest using kubectl apply -f app-deployment.yaml.

Rollback Conditions: Kubernetes provides various conditions under which a rollback can occur automatically. Some common rollback conditions include:

--rollback-to: You can manually trigger a rollback by specifying the previous revision to rollback to.

**--max-surge and --max-unavailable:** These options control the rate of change during a rollout. If the rollout fails, Kubernetes can use these parameters to determine whether to proceed with the rollout or initiate a rollback.

Monitor Rollout Status: Monitor the rollout status using kubectl rollout status deployment/app-deployment.

**Trigger Rollback:** If the rollout encounters issues or fails, Kubernetes will automatically trigger a rollback based on the configured conditions. If you need to manually trigger a rollback, you can use the kubectl rollout undo command, specifying the deployment name.

kubectl rollout undo deployment/app-deployment

**Verify Rollback:** After triggering the rollback, monitor the rollout status again to ensure that Kubernetes successfully rolls back to the previous version.

**6. Describe the differences between a ClusterIP service and a LoadBalancer service, providing a use case for each**

**ClusterIP Service:**

Description: A ClusterIP service exposes the application internally within the cluster using a virtual IP address. It is only accessible from within the cluster and is not exposed to the external network.

**Use Case:** A common use case for a ClusterIP service is when you have microservices architecture and want to enable communication between different services within the cluster. For example, if you have a frontend service that needs to communicate with a backend API service, you can create a ClusterIP service for the backend API to allow the frontend service to access it.

**LoadBalancer Service:**

Description: A LoadBalancer service exposes the application externally to the cluster by provisioning a load balancer from the cloud provider's infrastructure. It distributes incoming traffic across multiple backend pods and provides high availability and scalability.

**Use Case:** A typical use case for a LoadBalancer service is when you want to expose your application to the internet and distribute incoming traffic across multiple backend pods for scalability and fault tolerance. For example, if you have a web application that needs to be accessible from the internet, you can create a LoadBalancer service to expose it, allowing users to access the application from their browsers.

In summary, ClusterIP services are suitable for internal communication between services within the cluster, while LoadBalancer services are ideal for exposing applications to the external network, providing scalability and high availability.