Exercise Procedure 1: Deploying and Managing Pods

1. Deploying a Pod:

First, let's create a simple YAML file to define our pod. Create a file named `simple-pod.yaml` with the following contents:

```
apiVersion: v1
    kind: Pod
    metadata:
        name: simple-pod
    spec:
        containers:
            - name: nginx-container
            image: nginx:latest
```

Now, apply the YAML file to create the pod:

```
    kubectl apply -f simple-pod.yaml
```

Verify that the pod is running:

```
    kubectl get pods
```

You should see the `simple-pod` pod in the output with the status `Running`.

2. Viewing Pod Logs:

To view logs from the running pod, use the following command:

```
    kubectl logs simple-pod
```

This will display the logs from the `nginx-container` container within the `simple-pod` pod.

3. Accessing a Pod:

To access the shell inside the running pod, you can use the following command:

```
    kubectl exec -it simple-pod -- /bin/bash
```

This will open an interactive shell session inside the `simple-pod` pod. You can now run commands as if you were inside the container.

You have successfully deployed a pod, viewed its logs, and accessed its shell.

Exercise Procedure 2: Creating and Managing Deployments with Minikube

1. Create a Deployment:

Create a YAML file named `nginx-deployment.yaml` with the following contents to define 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-container
        image: nginx:latest
        ports:
        - containerPort: 80
```

Apply the YAML file to create the deployment:

```
kubectl apply -f nginx-deployment.yaml
```

Verify that the deployment is created:

```
kubectl get deployments
```

2. Scale Deployments:

Scale the number of replicas for the deployment to 5:

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

Verify that the deployment has been scaled:

```
kubectl get deployments
```

You should see the `nginx-deployment` with 5 replicas.

3. Update Deployments:

Update the deployment to use a different version of the Nginx image:

```
kubectl set image deployment/nginx-deployment nginx-container=nginx:1.21
```

Verify that the deployment is updated:

```
kubectl rollout status deployment/nginx-deployment
```

4. Deleting a Deployment (Optional):

Delete the deployment next:

```
kubectl delete deployment nginx-deployment
kubectl get deployments
```

5. Rolling Back Deployments (Optional):

If needed, you can roll back to the previous version of the deployment:

```
kubectl rollout undo deployment/nginx-deployment
```

Verify that the rollback is successful:

```
kubectl rollout status deployment/nginx-deployment
```

You have successfully created and managed deployments using Minikube on WSL Ubuntu 22.04.

Exercise Procedure 3: Managing Services with Minikube

1. Creating a Service:

Create a YAML file named `nginx-service.yaml` to define a service for the Nginx deployment:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

Apply the YAML file to create the service:

```
    kubectl apply -f nginx-service.yaml
```

Verify that the service is created:

```
    kubectl get services
```

2. Accessing Services:

To access the Nginx service, first, get the NodePort assigned to the service:

```
kubectl get svc nginx-service
```

Note the port number under the `PORT(S)` column.

Now, you can access the Nginx service using Minikube's IP address and the NodePort:

```
minikube ip
```

This command will give you the IP address of your Minikube cluster.

In a web browser, navigate to `http://<minikube-ip>:<node-port>`. Replace `<minikube-ip>` with the IP address from the previous command and `<node-port>` with the NodePort obtained from the `kubectl get svc nginx-service` command.

You should see the default Nginx welcome page if everything is set up correctly.

3. Cleaning Up (Optional):

If you no longer need the service, you can delete it using:

```
kubectl delete service nginx-service
```

You have successfully created and accessed a service for the Nginx deployment using Minikube on WSL Ubuntu 22.04.

Exercise Procedure 4: Using ConfigMaps and Secrets with Minikube

ConfigMaps in Kubernetes are key-value stores used to store non-sensitive configuration data, such as environment variables or configuration files, separate from application code. They enable developers to manage configuration settings independently of the application logic, promoting flexibility and ease of maintenance. By decoupling configuration from code, ConfigMaps facilitate streamlined application deployments and updates, allowing for dynamic configuration changes without the need for code modifications.

In contrast, Secrets provide a secure way to manage sensitive information, such as passwords, API keys, and TLS certificates, within Kubernetes clusters. Secrets are encrypted at rest and are base64-encoded, ensuring confidentiality and security. They play a crucial role in safeguarding sensitive data from unauthorized access or exposure, enabling organizations to adhere to compliance requirements and maintain data integrity in their containerized environments.

The requirement for ConfigMaps and Secrets arises from the need to effectively manage both configuration data and sensitive information in Kubernetes deployments. ConfigMaps allow for flexible configuration management, while Secrets ensure the secure handling of sensitive data. Together, ConfigMaps and Secrets empower organizations to deploy and manage containerized applications efficiently, balancing the need for configuration flexibility with the imperative to protect sensitive information and uphold data security standards.

1. Creating a ConfigMap:

Create a ConfigMap to store configuration data. For example, let's create a ConfigMap named `nginx-config` with a simple configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    server {
        listen 80;
        server_name localhost;

        location / {
            root /usr/share/nginx/html;
            index index.html;
        }
    }
```

Apply the ConfigMap to create it:

```
kubectl apply -f nginx-configmap.yaml
```

Verify that the ConfigMap is created:

```
kubectl get configmaps
```

2. Using ConfigMaps in Pods:

Now, let's create a pod that uses the ConfigMap. Create a YAML file named `nginx-pod-configmap.yaml` with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-configmap
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      volumeMounts:
        - name: config-volume
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
  volumes:
    - name: config-volume
      configMap:
        name: nginx-config
```

Apply the YAML file to create the pod:

```
kubectl apply -f nginx-pod-configmap.yaml
```

Verify that the pod is created:

```
kubectl get pods
```

3. Cleaning Up (Optional):

If you no longer need the resources created in this lab, you can delete them using:

```
kubectl delete configmap nginx-config
kubectl delete pod nginx-pod-configmap
```

You have successfully created and used ConfigMaps with Minikube on WSL Ubuntu 22.04.

Exercise Procedure 6: Using Persistent Volumes and Persistent Volume Claims with Minikube

Persistent Volumes (PVs) in Kubernetes serve as abstract representations of storage resources provisioned within the cluster, allowing for decoupling storage from individual pods. PVs can be backed by diverse storage types such as local storage, network-attached storage (NAS), or cloud-based solutions, providing flexibility in storage configurations. They enable administrators to manage storage resources independently of the applications using them, streamlining storage management tasks and ensuring efficient resource utilization.

Persistent Volume Claims (PVCs) are requests made by pods for storage resources in Kubernetes, abstracting away the complexities of storage provisioning and management. When a PVC is created, Kubernetes attempts to find an appropriate PV that matches the claim's requirements, facilitating automatic binding between the claim and the volume. PVCs enable developers to specify their storage needs without needing detailed knowledge of the underlying infrastructure, promoting simplicity and standardization in storage consumption across applications.

The requirement for Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) arises from the need to provide durable and scalable storage solutions for containerized applications in Kubernetes. By decoupling storage management from application deployment, PVs and PVCs offer a flexible and standardized approach to storage provisioning, enabling seamless data persistence and access. They ensure that applications have access to the required storage resources while allowing administrators to manage storage infrastructure efficiently, meeting the dynamic demands of modern containerized environments.

1. Creating a Persistent Volume:

Create a YAML file named `pv.yaml` to define a persistent volume with specific storage capacity and access modes:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

Apply the YAML file to create the persistent volume:

```
kubectl apply -f pv.yaml
```

Verify that the persistent volume is created:

```
kubectl get pv
```

2. Creating a Persistent Volume Claim:

Create a YAML file named `pvc.yaml` to define a persistent volume claim that requests storage from the persistent volume:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Apply the YAML file to create the persistent volume claim:

```
kubectl apply -f pvc.yaml
```

Verify that the persistent volume claim is created:

```
kubectl get pvc
```

3. Using Persistent Volumes in Pods:

Create a YAML file named `pod-pvc.yaml` to define a pod that mounts the persistent volume claim as a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      volumeMounts:
        - name: data-volume
          mountPath: /mnt/data
  volumes:
    - name: data-volume
      persistentVolumeClaim:
        claimName: my-pvc
```

Apply the YAML file to create the pod:

```
kubectl apply -f pod-pvc.yaml
```

Verify that the pod is created:

```
kubectl get pods
```

4. Cleaning Up (Optional):

If you no longer need the resources created in this lab, you can delete them using:

```
kubectl delete pv my-pv
kubectl delete pvc my-pvc
kubectl delete pod my-pod
```

You have successfully created and used persistent volumes and persistent volume claims with Minikube on WSL Ubuntu 22.04.

Exercise 7: Dashboard, Ingress

1. Starting the Dashboard:

- Start the Kubernetes dashboard

2. Enabling Ingress:

- Create an Ingress resource

3. Deleting an Ingress:

- Delete an existing Ingress

4. Listing Ingresses:

- To get a list of all Ingresses in all namespaces

Exercise Procedure 7: Dashboard, Ingress (kubernetes netoworking)

1. Starting the Dashboard:

- Start the Kubernetes dashboard
- The Kubernetes dashboard provides a graphical user interface (GUI) for managing and monitoring Kubernetes clusters. While it can be a helpful tool for visualizing resources and performing certain tasks, it's not always necessary for basic Kubernetes operations, especially when learning concepts through command-line interfaces (CLIs) like kubectl.
- we initially focused on using kubectl commands to perform various tasks such as deploying pods, creating services, managing persistent volumes, and configuring ingresses. These tasks can all be accomplished effectively using kubectl without the need for the dashboard.

```
minikube dashboard
```

2. Enabling Ingress:

- Create an Ingress resource
- an Ingress is an API object that manages external access to services running in a cluster. It provides HTTP and HTTPS routing capabilities to different services based on HTTP/HTTPS hostnames and paths. Essentially, an Ingress acts as a gateway or entry point to your Kubernetes cluster, allowing external traffic to reach services inside the cluster.
- Here's why Ingress is needed: Single Entry Point, HTTP(S) Routing, Load Balancing, TLS Termination and Centralized Configuration etc.

```
kubectl create ingress my-ingress --rule=host/path=my-service:port
```

3. Deleting an Ingress:

- Delete an existing Ingress

```
kubectl delete ingress my-ingress
```

4. Listing Ingresses:

- To get a list of all Ingresses in all namespaces

```
kubectl get ingress --all-namespaces
```

Exercise Procedure 8: Namespace Management and Cleanup

1. Creating a Namespace:

- Create a new namespace
- a namespace is a way to logically divide and isolate cluster resources into distinct groups. It provides a scope for objects such as pods, services, deployments, and replica sets within the cluster.
- Here's why namespaces are used: Resource Isolation, Multi-tenancy, Resource Quotas and Limits, Access Control, Organization and Management.

```
kubectl create namespace my-namespace
```

2. Changing Namespace:

- Change the current namespace

```
kubectl config set-context --current --namespace=my-namespace
```

3. Deleting a Namespace:

- Delete the namespace created earlier

```
kubectl delete namespace my-namespace
```

4. Listing Pods in a Namespace:

- To list pods running in a certain namespace:

```
kubectl get pods -n my-namespace
```

5. Deleting Pods, Services, and Deployments at Once:

- Delete all pods, services, and deployments at once:

```
kubectl delete pods,services,deployments --all
```

This lab covers namespace management and provides a way to clean up all resources at once.