Kubernetes Developers Labs

Overview

Lab 1: Getting Started with Containers

Running MinIO as a Container on Dockers

Lab 2: Kubernetes Fundamentals with Pods, ReplicaSets, and Services

Deploying MinIO as a Container on Kubernetes

Lab 3: Deploying MinIO as a Deployment with Persistent Volume in Kubernetes

Deploying MinIO as a Deployment with persistence enabled on Kubernetes

Lab 4: Upgrading MinIO as a StatefulSet in Kubernetes

Deploying MinIO as a StatefulSet

Lab 5: Managing Configuration with ConfigMaps and Secrets in Kubernetes

Configuring MinIO with ConfigMaps and Secrets in Kubernetes

Lab 6: Understanding and Creating Custom Resource Definitions (CRDs) in Kubernetes

Deploying MinIO as a CRD

Prerequisites

1. A pre-installed Docker environment
2. Docker Hub account
3. A pre-installed Kubernetes environment

Dockers & Kubernetes Labs

Lab 1: Getting started with Kubernetes

Running MinIO as a Container on Docker

Time: 20 Mins

Lab Summary: This lab introduces the fundamental concepts of containerization. Participants will be provided a docker pre-installed environment and they will pull a pre-built image from Docker Hub, and run a basic containerized application.

Objectives:

- Pulling a Pre-Built Image from Docker Hub
- Running a Containerized Application
- Exploring Container Management
- Removing Containers
- Creating your own docker image

Step 1: Pulling a Pre-Built Image from Docker Hub

1. **Open your terminal.**

   Ensure Docker is installed by running:

   ```
   docker --version
   ```

2. **Pull the MinIO image.**

   Pull the official MinIO image from Docker Hub

   ```
   docker pull minio/minio
   ```

3. **Verify the image is pulled.**

   Check if the image is available locally:

```
docker images
```

Step 2: Running a Containerized MinIO Application

1. **Run the MinIO container.**
   - Start a MinIO container with the following command.

```
(docker run -d -p 9000:9000 -p 9001:9001 --name minio  -e
"MINIO_ROOT_USER=admin"   -e "MINIO_ROOT_PASSWORD=password"
minio/minio server /data --console-address ":9001")
```

- **`docker run`**: This is the base command used to create and run a new container from a Docker image.
- **`-d`**:Runs the container in detached mode, meaning it runs in the background.
- **`-p 9000:9000`**: Maps port 9000 on the host machine to port 9000 in the container. Port 9000 is typically used by MinIO for S3 API access.
- **`-p 9001:9001`**:Maps port 9001 on the host machine to port 9001 in the container. Port 9001 is used for the MinIO web console.
- **`--name minio`**:Assigns the name `minio` to the container. This makes it easier to reference the container in future Docker commands.
- **`-e "MINIO_ROOT_USER=admin"`**:Sets the environment variable `MINIO_ROOT_USER` inside the container to `admin`. This environment variable defines the username for the MinIO root user.
- **`-e "MINIO_ROOT_PASSWORD=password"`**:Sets the environment variable `MINIO_ROOT_PASSWORD` inside the container to `password`. This environment variable defines the password for the MinIO root user.
- **`minio/minio`**:Specifies the Docker image to use, in this case, the official MinIO image from Docker Hub.
- **`server /data`**:This is the command that runs inside the container. It tells MinIO to start in server mode and use the `/data` directory inside the container as the storage location.
- **`--console-address ":9001"`**:This flag specifies the address and port where the MinIO web console will be accessible. Here, it's set to use port 9001.

2. **Check running containers.**
    - List all running containers to ensure MinIO is running. . Do you see your minio container has again started and is exposing on port 9000 and 9001. Port 9000 for accessing MinIO rest API and Port 9001 is to access MinIO console.

```
docker ps
```

3. **Access the MinIO Console.**
    - Open a web browser in your lab environment and go to `http://localhost:9001`.
    - Log in with the credentials `admin` (username) and `password` (password).

Step 3: Exploring Container Management

1. **View container logs.**

    Check the logs of the MinIO container to see its output

```
docker logs minio
```

2. **Access the container's shell (optional).**

    - If you need to explore the MinIO container's filesystem, you can start a shell session inside the container:

```
docker exec -it minio /bin/bash
```

    - After you are inside the container, run ls command

```
ls
```

    - To exit the container and to come back to terminal, run exit command

```
Exit
```

3. **Stop MinIO container.**

```
docker stop  minio
```

4. **Remove the MinIO container.**

After stopping the container rm the minio container

```
docker stop  minio
```

5. **Confirm Deletion**

Run docker ps command again to check whether container is still running or not

```
docker ps
```

Step 5: Creating Your Own Docker Image

1. **Create a simple Dockerfile.**
    - Create a new directory for your Dockerfile:

```
mkdir myminioapp
cd myminioapp
```

- Create a file named `Dockerfile` and open it in nano editor:

```
nano Dockerfile
```

- Add below content and press “Ctrl+X” and press ‘Y’ to save changes:

```
FROM minio/minio
CMD ["minio", "server", "/data", "--console-address",
":9001"]
```

1. **`FROM minio/minio`**:**Purpose**: This line specifies the base image for the Docker image you are building. The `FROM` instruction initializes a new build stage and sets the base image
2. **`CMD ["minio", "server", "/data"]`**:The `CMD` instruction specifies the command that will be executed when a container is started from this image. This is the default command that runs when no other command is provided during `docker run`.

2. **Build the Docker image.**
    - Build your image from the Dockerfile. This command will create an image named `myminioapp` with the `latest` tag.

```
docker build -t myminioapp:latest .
```

- Check your custom image

```
docker images
```

3. **Run the newly created image.**
    - Test your image by running a container. This time we are not providing the “”

```
(docker run -d -p 9000:9000 -p 9001:9001 --name myminioapp
-e "MINIO_ROOT_USER=admin"  -e
"MINIO_ROOT_PASSWORD=password"  myminioapp)
```

- Check your containers again. Do you see your minio container has again started and is exposing on port 9000 and 9001.

```
docker ps
```

4. **Access the MinIO Console.**

   Let's access the minio console Again.

- Open a web browser in your lab environment and go to `http://localhost:9001`.
- Log in with the credentials `admin` (username) and `password` (password).

Conclusion: In this lab, you learned the basics of working with containers using MinIO as an example. You pulled the MinIO image from Docker Hub, ran MinIO as a containerized application, explored container management, created your own Docker image. This foundational knowledge will help you leverage containerization for deploying and managing applications efficiently

---

END OF LAB

Lab 2: Kubernetes Fundamentals with Pods, ReplicaSets, and Services

Time: 25 mins

Deploying MinIO as a Container on Kubernetes

Lab Summary: In this lab, participants will deploy MinIO, a high-performance object storage solution, as a containerized application on Kubernetes. This lab introduces key Kubernetes concepts, including Pods, ReplicaSets, and Services, while guiding participants through the process of deploying MinIO. Starting with a basic Pod to run the MinIO container, participants will then create a ReplicaSet to ensure high availability and scalability. Finally, they will expose MinIO to external traffic using a Kubernetes Service, making it accessible for use. This lab is designed to reinforce the foundational Kubernetes concepts while providing practical experience with deploying real-world applications.

- Deploy MinIO as a Pod
- Ensure Availability with a ReplicaSet
- Expose MinIO Using a Service
- Test and Verify the Deployment

Step 1: Starting a Minikube Cluster

1. **Start Minikube with the specified resources.**

   Open your terminal and start a Minikube cluster with 2 CPUs and 4GB of memory:

   ```
   minikube start --cpus=2 --memory=4096
   ```

2. **Verify the Minikube cluster is running.**

   Ensure that the Minikube cluster is up and running:

   ```
   kubectl get nodes
   ```

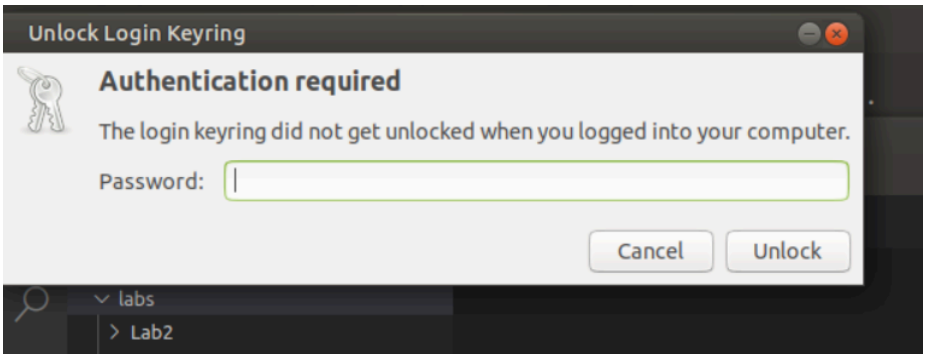
You should see a node with the status Ready.

Step 2: Open VS Code

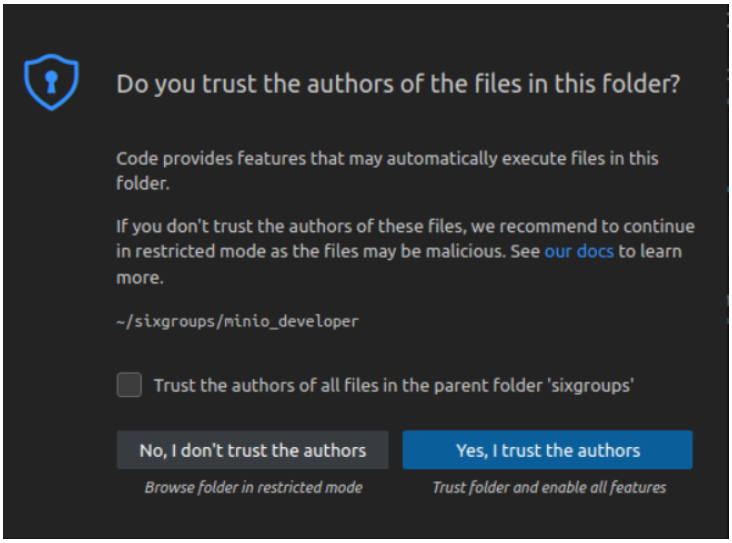
1. Open your kubernetes yaml discussed in all below labs in a VS Code. Run below to open labs in VS code.Open terminal and run below command to open VS Code.

```
cd sixgroups/kubernetes_developer/
Code .
```

- ***If you get below prompt, just press cancel***



- ***You might also get below prompt, just click on "yes, I trust the author "***



Step 3: Deploy MinIO as Pod

1. Under Lab2, you will find ***minio_pod.yaml***
2. Let's demystify this file.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: minio
  labels:
    app: minio
spec:
  containers:
  - name: minio
    image: minio/minio
    args:
    - server
    - /data
    env:
    - name: MINIO_ROOT_USER
      value: "admin"
    - name: MINIO_ROOT_PASSWORD
      value: "password"
    ports:
    - containerPort: 9000
    - containerPort: 9001
```

Step-by-Step Explanation:

```
1. apiVersion: v1:
Purpose: Specifies the API version used to create the resource.
Details: v1 is the stable API version for core Kubernetes objects like Pods,
Services, ConfigMaps, etc.

2. kind: Pod:
Purpose: Defines the type of Kubernetes resource you are creating.
Details: Pod is the smallest and simplest Kubernetes object, representing a
single instance of a running process in your cluster.

3. metadata::
Purpose: Provides metadata about the object, including its name and labels.
Details:
name: minio: Specifies the name of the Pod, which in this case is minio.
labels::
```

Labels are key-value pairs used to organize and select Kubernetes resources.
app: minio: This label categorizes the Pod under the app: minio label, which can be useful for selecting and grouping resources later.

4. spec::
Purpose: Defines the desired state and configuration of the Pod.
Details:
The spec section contains all the information needed to create and run the Pod, including the container specification.

5. containers::
Purpose: Lists the containers that will run inside the Pod.
Details:
A Pod can contain one or more containers. Here, there is only one container named minio.

- name: minio:
Purpose: Specifies the name of the container.
Details: This name is used within the Pod to identify the container.

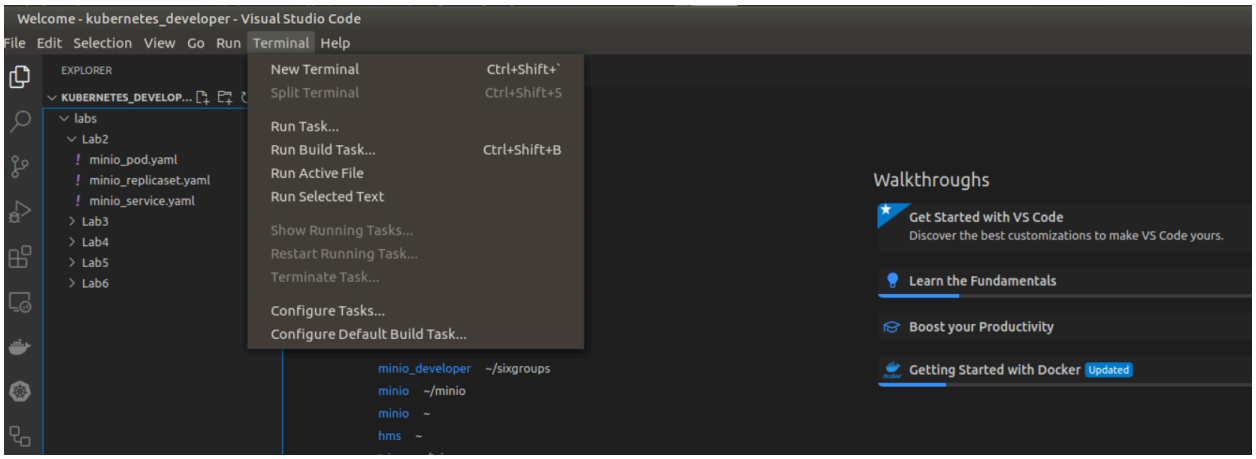
image: minio/minio:
Purpose: Specifies the Docker image to use for this container.
Details: The container will run the minio/minio image, which contains the MinIO server software.

args::
Purpose: Passes additional command-line arguments to the container's entry point.
Details:
- server: Instructs MinIO to run as a server.
- /data: Specifies the data directory where MinIO will store its data.

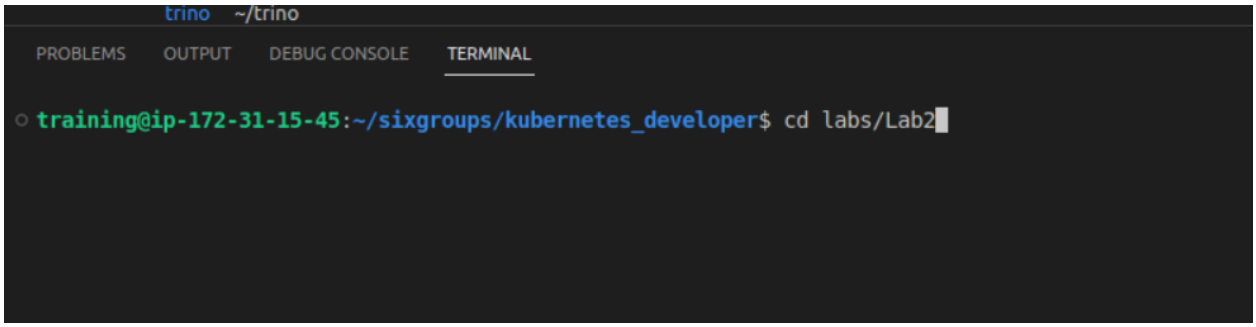
env::
Purpose: Defines environment variables for the container.
Details:
- name: MINIO_ROOT_USER: Sets the environment variable MINIO_ROOT_USER to admin, which is the username for the MinIO root user.
- name: MINIO_ROOT_PASSWORD: Sets the environment variable MINIO_ROOT_PASSWORD to password, which is the password for the MinIO root user.

ports::
Purpose: Exposes ports on the container to the network.
Details:
- containerPort: 9000: Exposes port 9000 for the MinIO API, allowing external access to the MinIO service.
- containerPort: 9001: Exposes port 9001 for the MinIO web console, allowing external access to the MinIO management interface.

2. Now open terminal inside VS code



3. **Go to labs/Lab2 folder**



4. **Apply the YAML file to create the Pod.**

   Deploy the Pod using the `kubectl apply` command:

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

5. **Verify that the Pod is running.**

   Check the status of the Pod:

```
kubectl get pods
```

```
training@ip-172-31-15-45:~/sixgroups/kubernetes_developer/labs/Lab2$ kubectl get pod
NAME    READY   STATUS    RESTARTS   AGE
minio   1/1     Running   0          10s
```

The `minio` Pod should show a status of `Running`.

6. **Delete the Pod**

   Check the status of the Pod:

```
kubectl delete pod minio
```

Step 3: Creating a ReplicaSet for MinIO

1. In the same Lab2 folder, we have YAML configuration for the MinIO ReplicaSet.

   This file defines a Kubernetes ReplicaSet named `minio-replicaset` that ensures three replicas of a Pod running the MinIO server are always up and running. Each Pod uses the `minio/minio` image, with ports `9000` and `9001` exposed, and environment variables set for the MinIO root user credentials.

   `minio_replicaset.yaml` with the following content:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: minio-replicaset
  labels:
    app: minio
spec:
  replicas: 3
  selector:
    matchLabels:
      app: minio
  template:
    metadata:
```

```
    labels:
      app: minio
  spec:
    containers:
    - name: minio
      image: minio/minio
      args:
      - server
      - /data
      env:
      - name: MINIO_ROOT_USER
        value: "admin"
      - name: MINIO_ROOT_PASSWORD
        value: "password"
      ports:
      - containerPort: 9000
      - containerPort: 9001
```

2. **Apply the YAML file to create the ReplicaSet.**

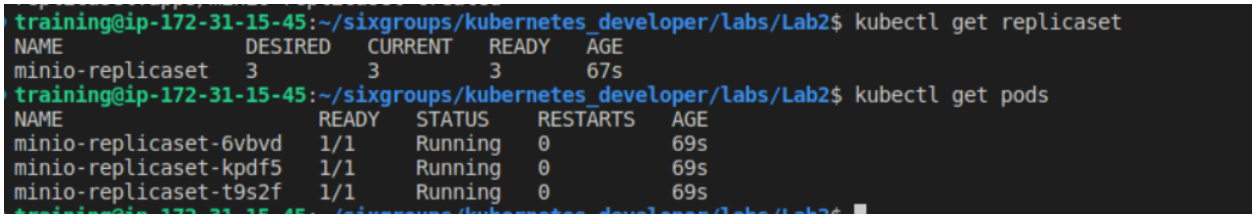
   Deploy the ReplicaSet using the `kubectl apply` command:
   `kubectl apply -f minio_replicaset.yaml`

3. **Verify that the ReplicaSet is functioning.**

   Check the status of the ReplicaSet and the number of running Pods:

```
kubectl get replicaset
kubectl get pods
```

You should see three Pods created by the ReplicaSet, all with the status `Running`.



Step 4: Exposing MinIO with a Kubernetes Service

1. In the same Lab2 folder, we have a YAML configuration for the MinIO Service.

   This YAML file defines a Kubernetes Service named `minio-service` that routes traffic to Pods with the `app: minio` label. It exposes ports `9000` and `9001` for the MinIO API and console, respectively, using the `NodePort` type to allow external access to these services.

   `minio_service.yaml` with the following content:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: minio-service
spec:
  selector:
    app: minio
  ports:
  - name: minio-svc
    protocol: TCP
    port: 9000
    targetPort: 9000
  - name: minio-console
    protocol: TCP
    port: 9001
    targetPort: 9001
  type: NodePort
```

2. **Apply the YAML file to create the Service.**

   Expose MinIO using the `kubectl apply` command:
   `kubectl apply -f minio_service.yaml`

3. **Check for services**
   `kubectl get services`

```
training@ip-172-31-15-45:~/sixgroups/kubernetes_developer/labs/Lab2$ kubectl get services
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)             AGE
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP             27m
minio-service   ClusterIP   10.102.29.109   <none>        9000/TCP,9001/TCP   43s
```

4. **Let's do port-forwarding for port 9001 to access MinIO Console.**

   This command forwards port `9001` on your local machine to port `9001` of the `minio-service` Kubernetes Service. This allows you to access the MinIO web console running inside the cluster from your local machine by navigating to `http://localhost:9001`

```
kubectl port-forward svc/minio-service 9001
```

`kubectl port-forward` is a temporary solution for accessing services within a Kubernetes cluster. It's mainly used for debugging, testing, or local development. The connection is only active while the command is running and is not suitable for production use, as it doesn't scale, and it only allows access from the machine running the command.

5. **Open the browser and access** http://localhost:9001 **to open minio-console.**

```
Username; admin
Password: password
```

Step 5: Testing and Scaling the Deployment

1. **Open a new terminal**

2. **Scale the ReplicaSet.**

   Increase the number of replicas to 5:

```
kubectl scale replicaset minio-replicaset --replicas=5
```

3. **Verify that two additional Pods are created:**

```
kubectl get pods
```

Step 6: Clean Up Resources

1. **Delete the resources:**

```
kubectl delete rs minio-replicaset
kubectl delete service minio-service
```

2. Verify all resources are deleted:

```
kubectl get all
```

Conclusion:

In this lab, you have deployed MinIO on Kubernetes using Pods, ReplicaSets, and Services. You started by deploying a single Pod, ensure high availability and scalability with a ReplicaSet, and exposed MinIO to external traffic using a Service. These foundational Kubernetes concepts are essential for deploying and managing real-world applications in a cloud-native environment.

END OF LAB

Lab 3: Deploying MinIO as a Deployment with Persistent Volume in Kubernetes

Time: 25 mins

Lab Summary

In this 25-minute lab, participants will deploy MinIO as a Deployment in Kubernetes, using Persistent Volumes (PVs) through an existing Storage Class to ensure data persistence. Although Deployments typically manage stateless applications, attaching a Persistent Volume enables MinIO to retain its state. The lab also covers upgrading the Deployment and performing rolling updates to ensure minimal downtime during updates.

Objectives

1. Create a Persistent Volume Claim using the existing Storage Class.
2. Deploy MinIO as a Deployment.
3. Expose MinIO using a Service.
4. Upgrade the Deployment and perform rolling updates.
5. Clean up resources.

Step 1: Create a Persistent Volume Claim (PVC) Using the Existing Storage Class

1. **Under Lab3, you have been provided with a YAML file for the Persistent Volume Claim:**

   Minio_pvc.yaml

   This YAML file defines a PersistentVolumeClaim (PVC) named `minio-pvc` that requests 1Gi of storage with `ReadWriteOnce` access, meaning it can be mounted as read-write by a single node. It uses the `standard` storage class to provision the requested storage in the Kubernetes cluster

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: minio-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

2. **In your VS code terminal go to labs/Lab3 folder**

3. **Apply the Persistent Volume Claim configuration:**

```
kubectl apply -f minio_pvc.yaml
```

3. **Verify the persistent volume created because of above claim**

```
kubectl get pv
```

Note: The STATUS of the PVC should be Bound, indicating that a Persistent Volume has been dynamically provisioned.

```
NAME        STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
minio-pvc   Bound    pvc-310b3ae5-f04a-4c09-add8-30bbd886bbf1   1Gi        RWO            standard       49s
```

Step 2: Deploy MinIO as a Deployment

1. **You have been provided with a YAML file for the MinIO Deployment in the same Lab3 folder:**

   ### Minio_deployment.yaml

   This YAML file defines a Kubernetes Deployment named `minio-deployment` that ensures 2 replicas of the MinIO server are running, each using the `minio/minio:latest` image. It mounts a PersistentVolumeClaim named `minio-pvc` to the `/data` directory in each container for persistent storage.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: minio-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: minio
  template:
    metadata:
      labels:
        app: minio
    spec:
      containers:
      - name: minio
        image: minio/minio:latest
        args:
        - server
        - /data
        env:
        - name: MINIO_ROOT_USER
          value: "admin"
        - name: MINIO_ROOT_PASSWORD
          value: "password"
        ports:
```

```
        - containerPort: 9000
          name: minio
        volumeMounts:
        - name: data
          mountPath: /data
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: minio-pvc
```

2. Make sure, you are in labs/Lab3 folder and then Apply the Deployment configuration:

```
kubectl apply -f minio_deployment.yaml
```

3. Verify the Deployment:

```
kubectl get deployment minio-deployment
```

4. Check the Pods:

```
kubectl get pods -l app=minio
```

Note: Ensure that the MinIO Pod is running before proceeding.

Step 3: Expose MinIO Using a Service

1. **In the same folder, you have a YAML file for the MinIO Service:**

   minio_service.yaml

It is the same file that we have discussed in a replica set as well. Since we deleted this service as part of cleanup, so we are going to create it again.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: minio-service
spec:
  selector:
    app: minio
  ports:
  - name: minio-svc
    protocol: TCP
    port: 9000
    targetPort: 9000
  - name: minio-console
    protocol: TCP
    port: 9001
    targetPort: 9001
  type: ClusterIP
```

2. Apply the Service configuration:

```
kubectl apply -f minio_service.yaml
```

3. Verify the Service:

```
kubectl get service minio-service
```

4. Let's forward 9001 port to access minio console on http://localhost:9001

```
kubectl port-forward svc/minio-service 9001
```

4. Open the browser and access http://localhost:9001

```
Username; admin
Password: password
```

Step 4: Upgrade the Deployment and Perform Rolling Updates

1. **Modify the Deployment YAML for an upgrade:**

   Open minio_deployment.yaml and Update the `image` field in the Deployment YAML to use a different version of MinIO

```
image: minio/minio:RELEASE.2024-08-26T15-33-07Z
```

2. **Apply the updated Deployment configuration:**

```
kubectl apply -f minio_deployment.yaml
```

3. **Monitor the rolling update:**

   Use the following command to monitor the rollout status:

```
kubectl rollout status deployment/minio_deployment
```

4. **Verify the Pods are updated**:

```
kubectl get pods -l app=minio
```

Step 5: Clean Up Resources

1. **Make sure you are in Lab3 folder, run below command to delete all the Deployment, PVC, and Service using yaml file**

```
kubectl delete -f .
```

2. Verify all resources are deleted:

```
kubectl get all
kubectl get pvc
```

Conclusion

Participants have successfully deployed MinIO as a Deployment in Kubernetes with Persistent Volumes to ensure data persistence. They also performed a rolling update to upgrade the MinIO deployment, ensuring minimal downtime during the update. This lab provides a hands-on experience in managing stateful applications using Kubernetes Deployments

END OF LAB

Lab 4: Upgrading MinIO as a StatefulSet in Kubernetes

Time: 25 mins

Lab Summary

In this lab, participants will deploy MinIO as a StatefulSet in Kubernetes, ensuring data persistence with Persistent Volumes. StatefulSets are ideal for stateful applications like MinIO, as they provide stable network identities, ordered deployment, and stable storage. This lab will guide participants through upgrading MinIO, explaining potential differences from deploying MinIO as a Deployment.

Objectives

1. Deploy MinIO as a StatefulSet.
2. Observe stable network identities.
3. Understand ordered deployment and termination.
4. Clean up resources.

Step 1: Deploy MinIO as a StatefulSet

Difference from Deployment: StatefulSets ensure that each Pod has a unique, persistent identity and storage, even across rescheduling. Unlike Deployments, StatefulSets are used for applications requiring stable network identities and persistent storage.

1. **Yaml for this lab is in Lab4 folder. You have been provided wuth a YAML file for the MinIO StatefulSet:**

   minio_statefulset.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: minio-statefulset
spec:
  serviceName: "minio"
  replicas: 2
  selector:
    matchLabels:
      app: minio
  template:
```

```
    metadata:
      labels:
        app: minio
    spec:
      containers:
      - name: minio
        image: minio/minio:latest
        args:
        - server
        - /data
        - --console-address
        - ":9001"
        env:
        - name: MINIO_ACCESS_KEY
          value: "admin"
        - name: MINIO_SECRET_KEY
          value: "password"
        ports:
        - containerPort: 9000
          name: minio
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
      storageClassName: standard
```

2. Apply the StatefulSet configuration:

```
kubectl apply -f minio-statefulset.yaml
```

3. Verify the StatefulSet

```
kubectl get statefulset minio-statefulset
```

4. Check the Pods:

```
kubectl get pods -l app=minio
```

Note: Each Pod in the StatefulSet will have a stable identifier, such as `minio-statefulset-0`

`minio-statefulset-1`

Step 2: Observe Stable Network Identities

Difference from Deployment: In StatefulSets, each Pod gets a consistent DNS name based on its ordinal index, ensuring stable network identities.

1. **Check the Pod DNS name:**

```
kubectl get pod minio-statefulset-0 -o jsonpath='{.spec.hostname}'
```

2. **Check the Pod's full DNS address:**

```
kubectl get pod minio-statefulset-0 -o jsonpath='{.status.podIP}'
```

- **The DNS name follows the pattern: `<pod_name>.<service_name>.<namespace>.svc.cluster.local`.**

Note: The stable network identity allows other applications to reliably connect to specific MinIO instances.

Step 3: Understand Ordered Deployment and Termination

Difference from Deployment: StatefulSets deploy and terminate Pods in an ordered fashion, ensuring that the next Pod only starts after the previous one is Running and Ready.

1. **Scale the StatefulSet to 3 replicas and Observe the ordered deployment:**

```
kubectl scale statefulset minio-statefulset --replicas=4
kubectl get pods -l app=minio --watch
```

Watch how the Pods are created one by one. To come out of this watch mode, press "Ctrl+c" or "Ctrl+x"

Note: Pods will be created in order (`minio-statefulset-0`, `minio-statefulset-1`, `minio-statefulset-2`, `minio-statefulset-3`).

Step 5: Clean Up Resources

1. **Delete the StatefulSet and PVC:**

```
kubectl delete statefulset minio-statefulset
kubectl delete pvc -l app=minio
```

END OF LAB

Lab 5: Managing Configuration with ConfigMaps and Secrets in Kubernetes

Time: 20 Mins

Lab Summary

In this lab, participants will enhance the MinIO StatefulSet by incorporating Kubernetes ConfigMaps and Secrets to manage configuration and sensitive information. They will learn how to externalize MinIO configuration using ConfigMaps and securely manage access keys using Secrets. This approach ensures that configuration and sensitive data are separated from the application code, making it easier to manage and secure stateful applications in Kubernetes.

Objectives

1. Create a ConfigMap for MinIO configuration.
2. Apply the ConfigMap using `kubectl`.
3. Verify the creation of the ConfigMap.
4. Create a Secret for MinIO access keys.
5. Apply the Secret using `kubectl`.
6. Verify the creation of the Secret.
7. Modify the MinIO StatefulSet to use ConfigMap and Secret.

Please Note: Yaml files for this lab are under labs/Lab5. Open terminal in vs code and navigate to that folder first

Step 1: Create a ConfigMap for MinIO Configuration

Purpose: ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable.

1. **In Lab5 folder, you have been provided with a YAML file for the MinIO ConfigMap:**

   minio_configmap.yaml

   This YAML file defines a ConfigMap named `minio-config` that stores configuration data for MinIO, including the volume path (`/data`) and the server URL (`http://minio:9000`). This configuration can be used to inject these settings into MinIO pod

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: minio-config
data:
  MINIO_VOLUMES: "/data"
  MINIO_SERVER_URL: "http://minio:9000"
```

2.Make sure you are in Lab5 folder before running below command.Apply the ConfigMap using kubectl:

```
kubectl apply -f minio_configmap.yaml
```

3.**Verify the creation of the ConfigMap**

```
kubectl get configmap minio-config -o yaml
```

Note: This ConfigMap will store MinIO configuration variables such as the storage volume location and the server URL.

Step 2: Create a Secret for MinIO Access Keys

Purpose: Secrets in Kubernetes allow you to store and manage sensitive information such as passwords, OAuth tokens, and SSH keys securely.

1. In the same Lab5 folder, you have been provided by a YAML file for the MinIO Secret:

   minio_secret.yaml

   This YAML file defines a Kubernetes Secret named minio-secret containing base64-encoded credentials for rootuser and rootpassword, which are minioadmin and miniosecret respectively. It is of type Opaque, used for storing arbitrary sensitive data.

```
apiVersion: v1
```

```
kind: Secret
metadata:
  name: minio-secret
type: Opaque
data:
  rootuser: bWluaW9hZG1pbg==  # Base64 encoded value for 'minioadmin'
  rootpassword: bWluaW9zZWNyZXQ=  # Base64 encoded value for 'miniosecret'
```

Note: The access and secret keys are base64 encoded.

2. **Apply the Secret using kubectl:**

```
kubectl apply -f minio_secret.yaml
```

3. **Verify the creation of the Secret:**

```
kubectl get secret minio-secret -o yaml
```

Note: Verify that the Secret is created and contains the expected data.

Step 3: Modify the MinIO StatefulSet to Use ConfigMap and Secret

Purpose: Modify the existing MinIO StatefulSet to use the newly created ConfigMap and Secret for its configuration and sensitive data.

`1. In the same lab5 folder, we have minio_statefulset_with_config.yaml file.:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: minio-statefulset
spec:
  serviceName: "minio"
  replicas: 1
  selector:
    matchLabels:
      app: minio
  template:
    metadata:
```

```
      labels:
        app: minio
    spec:
      containers:
      - name: minio
        image: minio/minio
        args:
        - server
        - /data
        - --console-address
        - ":9001"
        env:
        - name: MINIO_VOLUMES
          valueFrom:
            configMapKeyRef:
              name: minio-config
              key: MINIO_VOLUMES
        - name: MINIO_SERVER_URL
          valueFrom:
            configMapKeyRef:
              name: minio-config
              key: MINIO_SERVER_URL
        - name: MINIO_ROOT_USER
          valueFrom:
            secretKeyRef:
              name: minio-secret
              key: rootuser
        - name: MINIO_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: minio-secret
              key: rootpassword
        ports:
        - containerPort: 9000
          name: minio
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
```

```
      requests:
        storage: 1Gi
    storageClassName: standard
```

Note: In this modification:

The environment variables are sourced from the ConfigMap and Secret using env

2. Apply the updated StatefulSet:

```
kubectl apply -f minio_statefulset_with_config.yaml.yaml
```

3. Verify the StatefulSet configuration:

```
kubectl get statefulset
```

Note: Ensure that the StatefulSet is using the ConfigMap and Secret for its configuration and credentials.

Step 4: Verify the Configuration

1. **Check the Pods:**

   ```
   kubectl get pods -l app=minio
   ```

2. **Check the logs to confirm that MinIO is configured correctly:**

   ```
   kubectl logs minio-statefulset-0
   ```

3. Secrets and config variables are set as environment variables inside the pod.

Let's connect to our pod

```
kubectl exec -it minio-statefulset-0 /bin/bash
```

You will see below env variables inside your pod, notice "MINIO_ROOT_USER" and "MINIO_ROOT_PASSWORD", they are set from secret file.

MINIO_VOLUMES and MINIO_SERVER_URL are set from config file

```
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future versio
bash-5.1# env
MINIO_UPDATE_MINISIGN_PUBKEY=RWTx5Zr1tiHQLwG9keckT0c45M3AGeHD6IvimQHpyRywVWGbP1aV
KUBERNETES_SERVICE_PORT_HTTPS=443
MINIO_ROOT_PASSWORD=miniosecret
KUBERNETES_SERVICE_PORT=443
MINIO_SECRET_KEY_FILE=secret_key
HOSTNAME=minio-statefulset-0
MINIO_CONFIG_ENV_FILE=config.env
PWD=/
MINIO_VOLUMES=/data
MINIO_ROOT_USER_FILE=access_key
HOME=/root
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
MINIO_ACCESS_KEY_FILE=access_key
MINIO_ROOT_USER=minioadmin
TERM=xterm
SHLVL=1
KUBERNETES_PORT_443_TCP_PROTO=tcp
MINIO_KMS_SECRET_KEY_FILE=kms_master_key
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
MC_CONFIG_DIR=/tmp/.mc
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PORT=443
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MINIO_ROOT_PASSWORD_FILE=secret_key
MINIO_SERVER_URL=http://minio:9000
_=/usr/bin/env
```

Step 5: Clean Up Resources

```
kubectl delete statefulset minio-statefulset
kubectl delete configmap minio-config
kubectl delete secret minio-secret
```

END OF LAB

Lab 6: Understanding Custom Resource Definitions (CRDs) in Kubernetes and MinIO Operator

Time : 15 mins

Step 1: Introduction to Custom Resource Definitions (CRDs) in Kubernetes

What are Custom Resource Definitions (CRDs)?

Custom Resource Definitions (CRDs) are a powerful feature in Kubernetes that allows users to define their own custom resources. These resources function like the built-in resources in Kubernetes (such as Pods, Services, etc.) but are defined and managed according to user specifications. CRDs enable the extension of Kubernetes APIs, allowing you to manage custom applications and workflows within a Kubernetes cluster.

Why Use CRDs?

- **Extend Kubernetes API:** CRDs allow you to create new resource types that can be managed via the Kubernetes API.
- **Custom Workflows:** They enable the creation of custom workflows tailored to specific applications, such as managing databases, message queues, or any other service.
- **Automation:** Paired with custom controllers (often referred to as Operators), CRDs help automate complex operations, such as backup, restore, and scaling of applications.

1. **In the Lab6 folder, you have provided with a yaml where we are creating a sample CRD. Below yaml defines a new custom resource called `MyResource` under the group `mydomain.com`.**

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
```

```yaml
metadata:
  name: myresources.mydomain.com
spec:
  group: mydomain.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                field1:
                  type: string
  scope: Namespaced
  names:
    plural: myresources
    singular: myresource
    kind: MyResource
    shortNames:
    - mr
```

2. **Apply the CRD**

   First, apply the CRD to your Kubernetes cluster to create the new custom resource definition:

```
kubectl apply -f crd.yaml
```

3. **Check if the CRD has been successfully created:**

```
kubectl get crd
```

You should see `myresources.mydomain.com` listed among the CRDs.

4. In the same folder Lab6, there is another yaml file “myresource.yaml” .Let’s create an instance of your custom resource:

```yaml
apiVersion: mydomain.com/v1
kind: MyResource
metadata:
  name: example-myresource
spec:
  field1: "This is a test"
```

5. Save this YAML to a file, e.g., `myresource.yaml`, and apply it:

```
kubectl apply -f myresource.yaml
```

6. Verify the Instance

You can verify that the custom resource instance has been created by running:

```
kubectl get myresources
```

END OF LAB

Lab 7: Exploring MinIO Operator CRDs and Custom Controllers

Time: 25 Mins

What is an Operator?

An Operator is a special type of custom controller that is designed to manage complex applications. Operators use CRDs to define the application and its components, and they use the controller logic to manage the application lifecycle, including deployment, scaling, backup, recovery, and upgrades.

How Do CRDs and Custom Controllers Work Together?

- **CRDs:** Define the schema and API for a custom resource.
- **Custom Controller:** Watches for changes to instances of the CRD and takes actions to ensure that the actual state of the cluster matches the desired state defined in the CRD.

For example, if you define a CRD for a database and create an instance of this resource, the custom controller (or Operator) will ensure that the database is created, configured, and maintained according to the specifications in the CRD.

Step 3: Introduction to the MinIO Operator

Now that you have an understanding of CRDs and custom controllers, let's explore how the MinIO Operator leverages these concepts to manage MinIO deployments on Kubernetes.

What is the MinIO Operator?

The MinIO Operator is a Kubernetes Operator designed to simplify the deployment, management, and scaling of MinIO instances. It uses CRDs to define MinIO-specific resources, such as tenants, buckets, and consoles, and a custom controller to manage these resources.

Key Components:

- **CRDs:** Define custom resources like `Tenant`, `Bucket`, `Console`, and `Certificate`.
- **Custom Controller:** Automates the creation, management, and scaling of MinIO instances based on the custom resources.

Step 1: Run MinIO Operator using kubectl

```
kubectl apply -k "github.com/minio/operator?ref=v6.0.2"
```

Step 2: Verify operator pods are created in minio-operator namespace

```
kubectl get pod -n minio-operator
```

Step 3: Let's check what all CRD's are included by minio-operator

```
kubectl get crd -n minio-operator
```

Now we are going to create Tenant resource using tenant CRD found in our operator but before that let's create a namespace and secrets that is required in tenant yaml

Step 4: Let's first create a namespace with name "minio-tenant"

```
apiVersion: v1
kind: Namespace
metadata:
  name: minio-tenant
```

tep 4: Let's create two secrets to be used later in our Tenant CRD

```
apiVersion: v1
kind: Secret
metadata:
  name: storage-configuration
  namespace: minio-tenant
stringData:
  config.env: |-
    export MINIO_ROOT_USER="minio"
    export MINIO_ROOT_PASSWORD="minio123"
    export MINIO_STORAGE_CLASS_STANDARD="EC:2"
    export MINIO_BROWSER="on"
type: Opaque


---
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: storage-user
  namespace: minio-tenant
data:
  CONSOLE_ACCESS_KEY: Y29uc29sZQ==
  CONSOLE_SECRET_KEY: Y29uc29sZTEyMw==
type: Opaque
```

Step 5: Finally, let's create MinIO Tenant using tenant CRD spec.

Tenant Creation: When a `Tenant` resource is created, the MinIO Operator custom controller watches this resource and automatically creates the necessary Kubernetes resources (StatefulSets, Pods, PVCs, and Services) to deploy the MinIO instances.

```yaml
apiVersion: minio.min.io/v2
kind: Tenant
metadata:
  labels:
    app: minio
  name: myminio
  namespace: minio-tenant
spec:
  configuration:
    name: storage-configuration
  image: quay.io/minio/minio:RELEASE.2024-08-17T01-24-54Z
  mountPath: /export
  pools:
  - name: pool-0
    servers: 2
    volumeClaimTemplate:
      apiVersion: v1
      kind: persistentvolumeclaims
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 1Gi
```

```
      storageClassName: standard
  volumesPerServer: 2
requestAutoCert: true
users:
- name: storage-user
```

Above Tenant CRD Creates the Following Built-in Resources:

- **Pods: One Pod per MinIO server instance, each running a MinIO container.**
- **StatefulSets: A StatefulSet is created to manage the Pods, ensuring they have persistent storage and stable network identities.**
- **PersistentVolumeClaims (PVCs): PVCs are created for each storage volume defined in the `pools` section to provide persistent storage to the MinIO Pods.**
- **Services:**
    - **Headless Service: To manage the StatefulSet and provide stable network identities for the MinIO instances.**
    - **LoadBalancer/ClusterIP Service: To expose the MinIO service to external users.**
    - **Console Service: To expose the MinIO management console**

```
kubectl get tenant -n minio-tenant
```

Run the below command to see what all built-in objects are created

```
kubectl get all -n minio-tenant
Kubectl get pvc -n minio-tenant
```

END OF LAB