Project 01 - 1 Hour

Deploying a Scalable Web Application with Persistent Storage and Advanced Automation

Objective:

Deploy a scalable web application using Docker Swarm and Kubernetes, ensuring data persistence using a single shared volume, and automate the process using advanced shell scripting.

Overview:

1. **Step 1**: Set up Docker Swarm and create a service.
2. **Step 2**: Set up Kubernetes using Minikube.
3. **Step 3**: Deploy a web application using Docker Compose.
4. **Step 4**: Use a single shared volume across multiple containers.
5. **Step 5**: Automate the entire process using advanced shell scripting.

Step 1: Set up Docker Swarm and Create a Service

1.1 Initialize Docker Swarm

```
# Initialize Docker Swarm
docker swarm init
```

docker swarm join --token SWMTKN-1-3irx02mucesd6o4im18q5w0mt2qncx7jlp6zv1d6ee4703n8xe-2ww7asikjfo2ywkd037sejh8v 192.168.56.12:2377

```
vagrant@Master:~$ docker swarm --advertise-addr 192.168.56.12
unknown flag: --advertise-addr
See 'docker swarm --help'.

Usage:  docker swarm COMMAND

Manage Swarm

Commands:
  init        Initialize a swarm
  join        Join a swarm as a node and/or manager

Run 'docker swarm COMMAND --help' for more information on a command.

vagrant@Master:~$ docker swarm init --advertise-addr 192.168.56.12
Swarm initialized: current node (novdh8bb8ffhdho1ohzhm5ywm) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-3irx02mucesd6o4im18q5w0mt2qncx7jlp6zv1d6ee4703n8xe-2ww7asikjfo2ywkd03
7sejh8v 192.168.56.12:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

1.2 Create a Docker Swarm Service

```
# Create a simple Nginx service in Docker Swarm
docker service create --name nginx-service --publish 8080:80 nginx
```

```
vagrant@Master:~$ docker service create --name nginx-service --publish 8080:80 nginx
v4b83144i9baordpovzzc8jw2
overall progress: 1 out of 1 tasks
1/1: running
verify: Service v4b83144i9baordpovzzc8jw2 converged
vagrant@Master:~$
```

Step 2: Set up Kubernetes Using Minikube

2.1 Start Minikube

```
# Start Minikube
minikube start
```

```
vagrant@Master:~$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd
64
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 91.1M  100 91.1M    0     0   353k      0  0:04:24  0:04:24 --:--:--  339k
vagrant@Master:~$ sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd6
4
vagrant@Master:~$ minikube start
😄  minikube v1.33.1 on Ubuntu 22.04 (vbox/amd64)
✨  Automatically selected the docker driver. Other choices: ssh, none

❗  The requested memory allocation of 1963MiB does not leave room for system overhead (total system
memory: 1963MiB). You may face stability issues.
💡  Suggestion: Start minikube with less memory allocated: 'minikube start --memory=1963mb'

📌  Using Docker driver with root privileges
👍  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.44 ...
    > gcr.io/k8s-minikube/kicbase...:  1.61 KiB / 481.58 MiB  0.00% 2.68 KiB p/💾  Downloading Kubern
etes v1.30.0 preload ...
    > preloaded-images-k8s-v18-v1...:  342.90 MiB / 342.90 MiB  100.00% 159.25
    > gcr.io/k8s-minikube/kicbase...:  481.58 MiB / 481.58 MiB  100.00% 197.57
🔥  Creating docker container (CPUs=2, Memory=1963MB) ...
🐳  Preparing Kubernetes v1.30.0 on Docker 26.1.1 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔗  Configuring bridge CNI (Container Networking Interface) ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
💡  kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
vagrant@Master:~$
```

2.2 Deploy a Web App on Kubernetes

Create a deployment file named webapp-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3
  selector:
```

```
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: nginx
        ports:
        - containerPort: 80
```

Apply the deployment:

kubectl apply -f webapp-deployment.yaml

```
vagrant@Master:~$ sudo snap install kubectl--classic
error: cannot install "kubectl--classic": invalid instance name: invalid snap name:
       "kubectl--classic"
vagrant@Master:~$ sudo snap install kubectl --classic
kubectl 1.30.2 from Canonical✓ installed
vagrant@Master:~$ kubectl apply -f webapp-deployment.yaml
deployment.apps/webapp created
vagrant@Master:~$
```

2.3 Expose the Deployment

kubectl expose deployment webapp --type=NodePort --port=80

```
vagrant@Master:~$ kubectl expose deployment webapp --type=NodePort --port=80
service/webapp exposed
vagrant@Master:~$
```

Step 3: Deploy a Web Application Using Docker Compose

3.1 Create a **docker-compose.yml** File

```
version: '3'
services:
  web:
    image: nginx
    ports:
      - "8080:80"
    volumes:
      - webdata:/usr/share/nginx/html

volumes:
  webdata:
```

3.2 Deploy the Web Application

```
# Deploy using Docker Compose
docker-compose up -d
```

```
vagrant@Master:~$ vim docker-compose.yml
vagrant@Master:~$ docker compose up -d
WARN[0000] /home/vagrant/docker-compose.yml: `version` is obsolete
[+] Running 1/1
 ✔ Container vagrant-web-1  Started                                    0.7s
vagrant@Master:~$
```

Step 4: Use a Single Shared Volume Across Multiple Containers

4.1 Update **docker-compose.yml** to Use a Shared Volume

```
version: '3'
services:
  web1:
    image: nginx
    ports:
      - "8081:80"
    volumes:
      - shareddata:/usr/share/nginx/html
  web2:
    image: nginx
    ports:
      - "8082:80"
    volumes:
      - shareddata:/usr/share/nginx/html

volumes:
  shareddata:
```

4.2 Deploy with Docker Compose

```
# Deploy using Docker Compose
docker-compose up -d
```

```
vagrant@Master:~$ docker compose up -d
WARN[0000] /home/vagrant/docker-compose.yml: `version` is obsolete
WARN[0000] Found orphan containers ([vagrant-web-1]) for this project. If you removed or renamed this
 service in your compose file, you can run this command with the --remove-orphans flag to clean it up
.
[+] Running 3/3
 ✔ Volume "vagrant_shareddata"  Created                                0.0s
 ✔ Container vagrant-web2-1     Started                                0.6s
 ✔ Container vagrant-web1-1     Started                                0.7s
vagrant@Master:~$
```

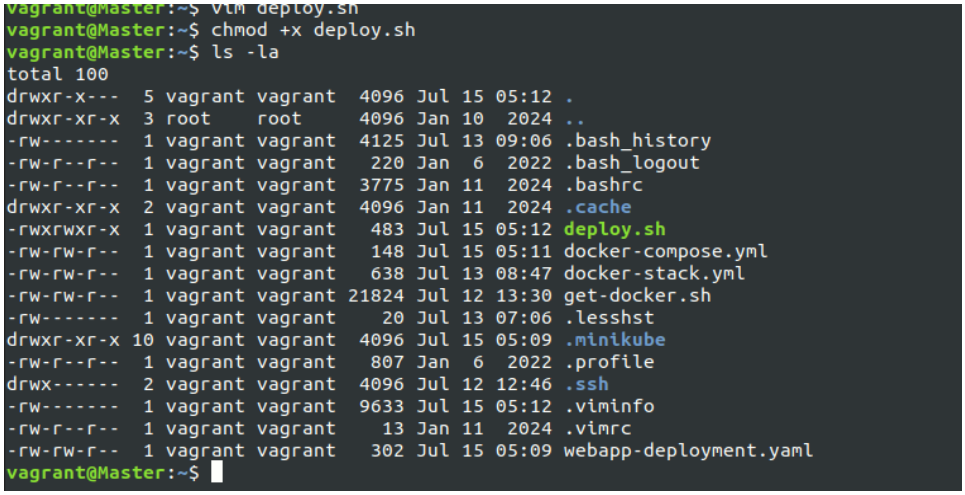
Step 5: Automate the Entire Process Using Advanced Shell Scripting

5.1 Create a Shell Script **deploy.sh**

```
#!/bin/bash

# Initialize Docker Swarm
docker swarm init

# Create Docker Swarm Service
docker service create --name nginx-service --publish 8080:80 nginx

# Start Minikube
minikube start

# Create Kubernetes Deployment
kubectl apply -f webapp-deployment.yaml

# Expose the Deployment
kubectl expose deployment webapp --type=NodePort --port=80

# Deploy Web App Using Docker Compose
docker-compose -f docker-compose-single-volume.yml up -d

echo "Deployment completed successfully!"
```

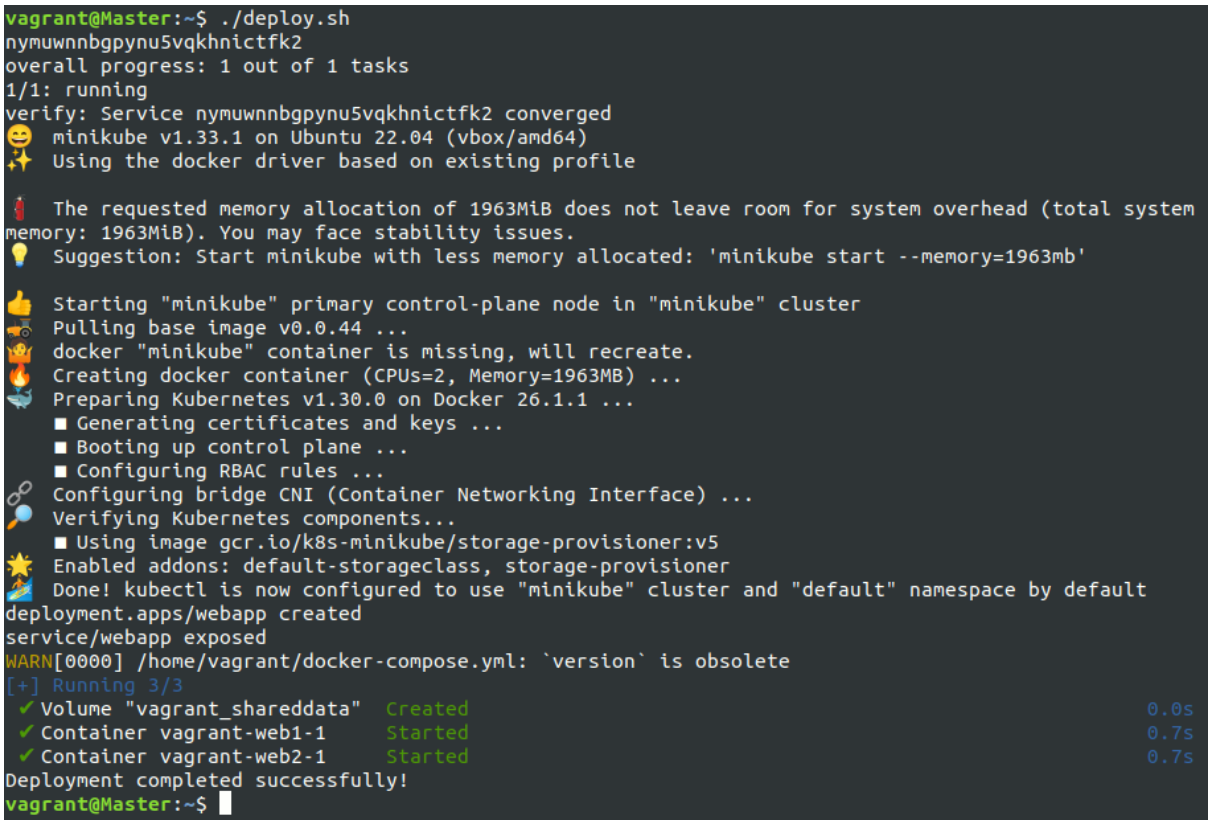
5.2 Make the Script Executable

```
# Make the script executable
chmod +x deploy.sh
```



5.3 Run the Script

```
# Run the deployment script
./deploy.sh
```



Project 02 - 1 Hour

Comprehensive Deployment of a Multi-Tier Application with CI/CD Pipeline

Objective:

Deploy a multi-tier application (frontend, backend, and database) using Docker Swarm and Kubernetes, ensuring data persistence using a single shared volume across multiple containers, and automating the entire process using advanced shell scripting and CI/CD pipelines.

Overview:

1. **Step 1**: Set up Docker Swarm and create a multi-tier service.
2. **Step 2**: Set up Kubernetes using Minikube.
3. **Step 3**: Deploy a multi-tier application using Docker Compose.

4. **Step 4**: Use a single shared volume across multiple containers.
5. **Step 5**: Automate the deployment process using advanced shell scripting.

Step 1: Set up Docker Swarm and Create a Multi-Tier Service

1.1 Initialize Docker Swarm

```
# Initialize Docker Swarm
docker swarm init
```

1.2 Create a Multi-Tier Docker Swarm Service

Create a docker-compose-swarm.yml file:

```
version: '3.7'
services:
  frontend:
    image: nginx
    ports:
      - "8080:80"
    deploy:
      replicas: 2
    volumes:
      - shareddata:/usr/share/nginx/html
  backend:
    image: mybackendimage
    ports:
      - "8081:80"
    deploy:
      replicas: 2
    volumes:
      - shareddata:/app/data
  db:
    image: postgres
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    deploy:
      replicas: 1
    volumes:
      - dbdata:/var/lib/postgresql/data

volumes:
  shareddata:
  dbdata:
```

Deploy the stack:

```
# Deploy the stack using Docker Swarm
docker stack deploy -c docker-compose-swarm.yml myapp
```

```
vagrant@Master:~/newProject$ docker stack deploy -c docker-compose-swarm.yml myapp
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
Creating network myapp_default
Creating service myapp_backend
Creating service myapp_db
Creating service myapp_frontend
vagrant@Master:~/newProject$
```

Step 2: Set up Kubernetes Using Minikube

2.1 Start Minikube

```
# Start Minikube
minikube start
```

```
vagrant@Master:~/newProject$ minikube start
😄  minikube v1.33.1 on Ubuntu 22.04 (vbox/amd64)
✨  Using the docker driver based on existing profile

❗  The requested memory allocation of 1963MiB does not leave room for system overhead (total system
memory: 1963MiB). You may face stability issues.
💡  Suggestion: Start minikube with less memory allocated: 'minikube start --memory=1963mb'

👍  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.44 ...
🤷  docker "minikube" container is missing, will recreate.
🔥  Creating docker container (CPUs=2, Memory=1963MB) ...
❗  This container is having trouble accessing https://registry.k8s.io
💡  To pull new external images, you may need to configure a proxy: https://minikube.sigs.k8s.io/docs
/reference/networking/proxy/
🐳  Preparing Kubernetes v1.30.0 on Docker 26.1.1 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔗  Configuring bridge CNI (Container Networking Interface) ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
vagrant@Master:~/newProject$
```

2.2 Create Kubernetes Deployment Files

Create frontend-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
```

```
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: shareddata
          mountPath: /usr/share/nginx/html
      volumes:
      - name: shareddata
        persistentVolumeClaim:
          claimName: shared-pvc
```

Create backend-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: mybackendimage
        ports:
        - containerPort: 80
        volumeMounts:
        - name: shareddata
          mountPath: /app/data
      volumes:
```

```
      - name: shareddata
        persistentVolumeClaim:
          claimName: shared-pvc
```

Create db-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
      - name: db
        image: postgres
        env:
        - name: POSTGRES_DB
          value: mydb
        - name: POSTGRES_USER
          value: user
        - name: POSTGRES_PASSWORD
          value: password
        volumeMounts:
        - name: dbdata
          mountPath: /var/lib/postgresql/data
      volumes:
      - name: dbdata
        persistentVolumeClaim:
          claimName: db-pvc
```

Create shared-pvc.yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: shared-pvc
spec:
  accessModes:
```

```
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Create db-pvc.yaml:

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

Apply the deployments:

kubectl apply -f shared-pvc.yaml

```
vagrant@Master:~/newProject$ vim shared-pvc.yaml
vagrant@Master:~/newProject$ kubectl apply -f shared-pvc.yaml
persistentvolumeclaim/shared-pvc created
vagrant@Master:~/newProject$
```

kubectl apply -f db-pvc.yaml

```
vagrant@Master:~/newProject$ kubectl apply -f db-pvc.yaml
persistentvolumeclaim/db-pvc unchanged
vagrant@Master:~/newProject$
```

kubectl apply -f frontend-deployment.yaml
kubectl apply -f backend-deployment.yaml

```
vagrant@Master:~/newProject$ kubectl apply -f frontend-deployment.yaml
deployment.apps/frontend created
vagrant@Master:~/newProject$ kubectl apply -f backend-deployment.yaml
deployment.apps/backend created
```

kubectl apply -f db-deployment.yaml

```
vagrant@Master:~/newProject$ kubectl apply -f db-deployment.yaml
persistentvolumeclaim/shared-pvc configured
vagrant@Master:~/newProject$
```

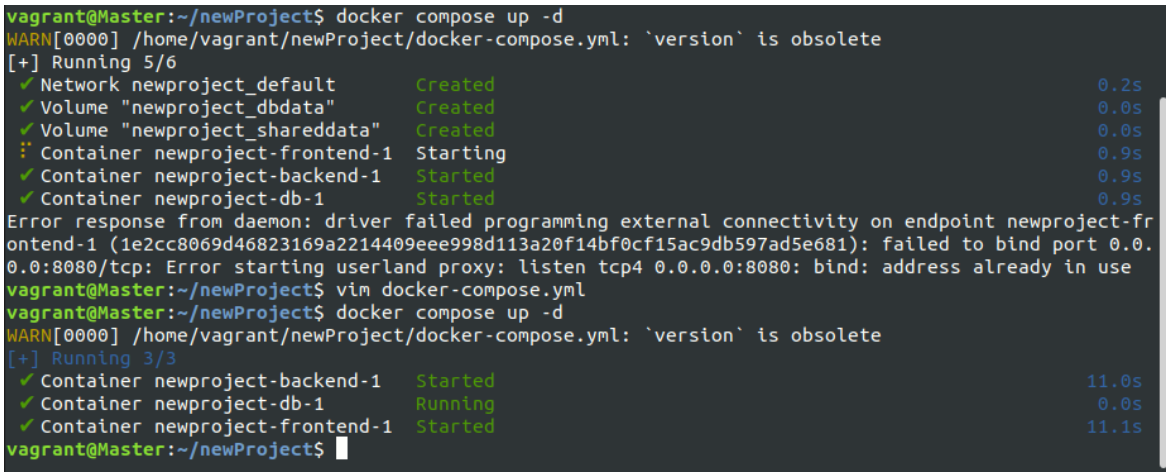
Step 3: Deploy a Multi-Tier Application Using Docker Compose

3.1 Create a **docker-compose.yml** File

```
version: '3'
services:
  frontend:
    image: nginx
    ports:
      - "8080:80"
    volumes:
      - shareddata:/usr/share/nginx/html
  backend:
    image: mybackendimage
    ports:
      - "8081:80"
    volumes:
      - shareddata:/app/data
  db:
    image: postgres
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - dbdata:/var/lib/postgresql/data

volumes:
  shareddata:
  dbdata:
```

3.2 Deploy the Application

```
# Deploy using Docker Compose
docker-compose up -d
```



Step 4: Use a Single Shared Volume Across Multiple Containers

Update `docker-compose.yml` as shown in Step 3.1 to use the `shareddata` volume across the frontend and backend services.