Deploying a Python Server Application using Kubernetes on Docker Desktop

This guide will walk you through deploying a **Flask-based Python server** using **Kubernetes** on **Docker Desktop**. You'll **containerize the application** using Docker, deploy it as a **Kubernetes Deployment**, and expose it using a **NodePort service** so that it is accessible over the network.

Why Are We Doing This?

Before jumping into implementation, let's understand why each step is necessary:

1 Why Docker and Kubernetes?

- **Docker** helps us package our Python server into a container so that it can run consistently across different environments.
- **Kubernetes** helps us **orchestrate** and **manage** multiple instances (pods) of our server, providing scalability and fault tolerance.

2 Why Use NodePort?

- A **NodePort service** exposes our application to external users by assigning it a port on the Kubernetes node.
- This allows us to access the application from any machine in the network using **Docker-Desktop-IP:NodePort**.

Step-by-Step Implementation

Let's now implement this step by step.

Step 1: Verify Docker and Kubernetes Are Running

Since our application is deployed using Kubernetes, we must ensure Kubernetes is running on Docker Desktop.

1.1 Verify Kubernetes is Running

Run the following command to check the cluster status:

```
kubectl cluster-info
```

If Kubernetes is working correctly, you should see information about the control plane.

1.2 Verify Docker Installation

Check the Docker installation:

```
docker --version
```

Expected Output:

```
Docker version 24.x.x, build xxxxxxx
```

If Kubernetes is not running, enable it from **Docker Desktop → Settings → Kubernetes → Enable Kubernetes**.

Step 2: Create a Python Server Application

We will create a simple **Flask-based Python application**.

2.1 Create a Project Directory

Navigate to your workspace and create a new directory:

```
mkdir python-k8s-app && cd python-k8s-app
```

2.2 Create `app.py` (Python Server)

Create a Python file (`app.py`) with the following content:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, Kubernetes with NodePort!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80)
```

Why? - **Flask** is a lightweight web framework for Python. - The `home()` function handles requests to `/` and returns a message. - The application listens on port `80`, making it container-friendly.

Step 3: Containerize the Application using Docker

To run this app inside a Kubernetes cluster, we need to **containerize** it using Docker.

3.1 Create a `Dockerfile`

Inside `python-k8s-app/`, create a `Dockerfile`:

```
# Use an official Python runtime as a parent image
FROM python:3.9

# Set the working directory in the container
WORKDIR /app

# Copy the application files into the container
COPY app.py /app

# Install Flask
RUN pip install flask

# Expose the application port
EXPOSE 80

# Run the Python server
CMD ["python", "app.py"]
```

Why? - Uses `python:3.9` as the base image. - Copies `app.py` inside the container. - Installs Flask inside the container. - Exposes port `80` (same as in `app.py`). - Runs `app.py` on startup.

3.2 Build the Docker Image

Run the following command to **build the image**:

```
docker build -t my-app .
```

Verify the image:

```
docker images
```

Expected Output:

```
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
my-app       latest    abcdef123456   10 seconds ago 50MB
```

3.3 Test the Docker Image Locally

Before deploying to Kubernetes, test the image:

```
docker run -p 4000:80 my-app
```

Now, open a browser and visit:

`http://localhost:4000`

Expected output:

```
Hello, Kubernetes with NodePort!
```

Step 4: Load the Docker Image into Kubernetes

Since Kubernetes doesn't access local Docker images by default, we need to load it properly.

4.1 Use a Private Registry (Alternative)

If you have multiple nodes, push the image to a local registry:

```
docker run -d -p 5000:5000 --name registry registry:2
docker tag my-app:latest localhost:5000/my-app:latest
docker push localhost:5000/my-app:latest
```

Step 5: Deploy the Application in Kubernetes

Now, we will create **Kubernetes deployment and service** files.

5.1 Create `deployment.yaml` (3 Replicas)

Create `deployment.yaml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: localhost:5000/my-app:latest
```

```
        ports:
        - containerPort: 80
```

Why? - Deploys **3 replicas** for redundancy. - Uses the image from our local registry. - Exposes container port `80`.

5.2 Create `service.yaml` (NodePort Service)

Create `service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30000
```

Why? - Uses **NodePort** to expose the service. - Assigns **port 30000** on the node.

Step 6: Deploy the Application

Now, apply the configurations.

6.1 Deploy to Kubernetes

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

6.2 Verify Deployment

Check running pods:

```
kubectl get pods
```

Expected Output:

```
NAME             READY   STATUS    RESTARTS   AGE
my-app-xyz123    1/1     Running   0          30s
```

```
my-app-abc456    1/1     Running   0          30s
my-app-mno789    1/1     Running   0          30s
```

6.3 Check Running Services

```
kubectl get services
```

Expected Output:

```
NAME             TYPE       CLUSTER-IP      EXTERNAL-IP
PORT(S)        AGE
my-app-service   NodePort   10.96.0.1       <none>
80:30000/TCP   30s
```

Step 7: Access the Application

Find your **Docker Desktop IP**:

```
kubectl get nodes -o wide
```

Use curl to access the service:

```
curl http://192.168.1.100:30000
```

Expected Output:

```
Hello, Kubernetes with NodePort!
```

Step 8: Cleanup (Optional)

To remove everything:

```
kubectl delete -f deployment.yaml
kubectl delete -f service.yaml
```

Summary

You have successfully: Deployed a Python server on Kubernetes
Containerized it with Docker
Exposed it using NodePort
Verified that it works

Now, your application is **scalable, containerized, and accessible from any machine**.