DevOps Lab:

Docker Container Sample using Node JS Calculator Webapp:

Step 1: Project Structure

```
calculator-webapp/
├── src/
│   └── index.js
├── public/
│   └── index.html
├── package.json
└── Dockerfile
```

Step 2: Updated **index.js** (located in src/index.js):

```
const express = require('express');
const app = express();
const port = 3000;

// Serve static files (like HTML, CSS)
app.use(express.static('public'));

// Basic routes for calculator operations
app.get('/add', (req, res) => {
   const { a, b } = req.query;
   const result = parseFloat(a) + parseFloat(b);
   res.send(`Result: ${result}`);
});

app.get('/subtract', (req, res) => {
   const { a, b } = req.query;
   const result = parseFloat(a) - parseFloat(b);
   res.send(`Result: ${result}`);
});
```

```
app.get('/multiply', (req, res) => {
    const { a, b } = req.query;
    const result = parseFloat(a) * parseFloat(b);
    res.send(`Result: ${result}`);
});

app.get('/divide', (req, res) => {
    const { a, b } = req.query;
    if (parseFloat(b) !== 0) {
        const result = parseFloat(a) / parseFloat(b);
        res.send(`Result: ${result}`);
    } else {
        res.send('Error: Division by zero');
    }
});

app.listen(port, () => {
    console.log(`Calculator app listening at http://localhost:${port}`);
});
```

Step 3: Updated **package.json**

```
{
  "name": "calculator-webapp",
  "version": "1.0.0",
  "description": "A simple web calculator application using Node.js and Express",
  "main": "src/index.js",
  "scripts": {
    "start": "node src/index.js"
  },
  "dependencies": {
```

```
    "express": "^4.17.1"
  },
  "author": "",
  "license": "ISC"
}
```

Step 4: Dockerfile

```
# Use the official Node.js image as the base image
FROM node:14
# Set the working directory
WORKDIR /usr/src/app
# Copy package.json and package-lock.json files
COPY package*.json ./
# Install the app dependencies
RUN npm install
# Copy the rest of the application code
COPY . .
# Expose the port that the app runs on
EXPOSE 3000
# Define the command to run the application
CMD ["node", "src/index.js"]
```

Step 5: Build the Docker Image

We will now build the Docker image from our Dockerfile.

```
docker build -t calculator-webapp .
```

we can inspect the image version using the below command:

docker inspect calculator-webapp (or) *docker inspect <image-name>*

Step 6: Run the Docker Container

Let's run our newly built Docker container.

```
docker run -d -p 3000:3000 --name gnit-calculator calculator-webapp
```

To Remove the container, we use below commands:

```
docker stop <container-id>
docker rm <container-id>
```

To Remove the image, we use below commands:

docker rmi <image-name>

Now let's automate running the above containerized Node.js calculator webapp using Kubernetes

Kubernetes way of Deployment

Step 1: We'll need a few configuration files to define your Kubernetes deployment and service.

Deployment Configuration (deployment.yml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: calculator-webapp-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: calculator-webapp
  template:
    metadata:
      labels:
        app: calculator-webapp
    spec:
      containers:
      - name: calculator-webapp
        image: calculator-webapp:latest
        ports:
        - containerPort: 3000
```

Service Configuration (service.yml)

```
apiVersion: v1
kind: Service
metadata:
  name: calculator-webapp-service
spec:
  selector:
    app: calculator-webapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: LoadBalancer
```

Note: As we are running Kubernetes on a local setup like Minikube or a cluster that doesn't support LoadBalancer services by default, we might need to use a different method, such as NodePort or using an ingress controller.

Service Configuration(service_nodeport.yml)

```
apiVersion: v1
kind: Service
metadata:
  name: calculator-webapp-service
spec:
  selector:
    app: calculator-webapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
      nodePort: 30001
  type: NodePort
```

Step 2: Build and Push Docker Image

```
        # Build the Docker image
```

docker build -t your_dockerhub_username/calculator-webapp:latest .

Note: Make sure to build the image with the above command

 # Push the Docker image to Docker Hub

docker push your_dockerhub_username/calculator-webapp:latest

Make sure to replace **your_dockerhub_username** with your **actual Docker Hub username**.

Step 3: Make sure your Kubernetes is running using the below commands:

set NO_PROXY=localhost,127.0.0.1,10.96.0.0/12,192.168.59.0/24,192.168.49.0/24,192.168.39.0/24

minikube start

Step 4: Deploy to Kubernetes

Use kubectl to deploy your application to a Kubernetes cluster.

 # Apply the deployment configuration

kubectl apply -f deployment.yml

 # Apply the service configuration

kubectl apply -f service.yaml

 # Apply the node port configuration (incase of local kubectl)

kubectl apply -f service_nodeport.yaml

Step 5: Verify the Deployment

Check the status of your pods and service to ensure everything is running smoothly.

 # Get the status of pods

kubectl get pods

 # Get the status of the service

kubectl get services

Step 6: Access Your Application

The LoadBalancer service will provide an external IP address that you can use to access your application. Run the following command to find out the external IP:

kubectl get services (or) kubectl describe service calculator-webapp-service

kubectl get services -o wide

Look for the EXTERNAL-IP column corresponding to the calculator-webapp-service. In this case we will CLUSTER-IP column for the webapp-service.

Run *minikube ssh*

Then run curl <cluster-IP> of the nodeport service.

Incase if we are still unable to open the service URLs, run the below commands:

minikube service calculator-webapp-service