Comprehensive Deployment of a Node.js Application with Docker, Jenkins, Kubernetes, Volumes, and Networks

Proficient Assessment

Contents

Title: Comprehensive Deployment of a Node.js Application with Docker, Jenkins, Kubernetes, Volumes, and Networks. 3
Difficulty Level 3
What You Will Learn 3
What You Need to Know 3
Skill Tags 3
What you will do 4
Activities 4
Step 1: Set Up the Environment: 4
Step 2: Navigate the Node.js Application Repository: 4
Step 3: Create and Configure a Dockerfile: 4
Step 4: Build and Tag the Docker Image: 4
Step 5: Push the Docker Image to Docker Hub: 5
Step 6: Set Up Jenkins for CI/CD: 5
Step 7: Deploy the Application on Kubernetes: 5
Step 8: Test the Kubernetes Deployment: 5
Step 9: Add a Persistent Volume and Persistent Volume Claim 6
Step 10: Create a Network Policy to Allow Traffic from Specific Pods 6
Testcases 7

Title: Comprehensive Deployment of a Node.js Application with Docker, Jenkins, Kubernetes, Volumes, and Networks.

Duration

• 180 minutes

Difficulty Level

• Proficient

What You Will Learn

- How to clone a Node.js application repository and set up a development environment.
- Creating and configuring a Dockerfile to containerize applications.
- Building, tagging, and pushing Docker images to Docker Hub.
- Setting up Jenkins for CI/CD pipelines to automate builds and deployments.
- Creating Kubernetes deployments and services for scalable applications.
- Managing Docker networks and Kubernetes volumes for persistent storage and inter-container communication.
- Testing and validating application deployments using Minikube.
- Understanding volume persistence and Kubernetes networking.

What You Need to Know

- Basic knowledge of Docker, including image building, tagging, and pushing to repositories.
- Familiarity with Kubernetes concepts like deployments, services, and volumes.
- Understanding of Jenkins pipelines and its plugins for CI/CD workflows.
- Basic Git usage for cloning repositories and navigating projects.
- Command-line interface experience for Docker, Kubernetes (kubectl), and Jenkins setup.
- Familiarity with Minikube for local Kubernetes cluster management.

Skill Tags

- DevOps

- Containerization
- Orchestration
- Automation
- Networking
- Testing and Validation

What you will do

In this lab, you will containerize a Node.js application using Docker, set up a CI/CD pipeline with Jenkins, and deploy the application to a Kubernetes cluster. You will configure Docker networks, manage Kubernetes volumes for persistent storage, and test the deployment using Minikube. You will also implement persistent storage, create network policies to control pod communication, and test the entire setup for functionality and security.

Activities

Step 1: Set Up the Environment:

1. Verify that Git, Docker, Minikube, and Jenkins are installed and properly configured on your system.
2. Ensure that Minikube is started, and kubectl is pointing to the correct context (minikube).

Step 2: Navigate the Node.js Application Repository:

1. Navigate to **"node-js-sample"** folder the **Project** folder present on **Desktop**.

Step 3: Create and Configure a Dockerfile:

1. Inside the project directory, create a **Dockerfile** to containerize the Node.js application.
2. Highlight key configurations like **EXPOSE 5000** and the **CMD** to start the application.

Step 4: Build and Tag the Docker Image:

1. Build a Docker image from the Dockerfile with the name **"node-js-sample "**and tag **"1.0"**.
   docker build -t node-js-sample:1.0 .

Step 5: Push the Docker Image to Docker Hub:

1. Log in to your Docker Hub account:
   docker login

2. Tag the image with your Docker Hub username:
   docker tag node-js-sample:1.0 <your-dockerhub-username>/node-js-sample:1.0

3. Push the image to Docker Hub:
   docker push <your-dockerhub-username>/node-js-sample:1.0

Step 6: Set Up Jenkins for CI/CD:

1. Access Jenkins and create a new pipeline job named **"Node.js-Deployment"**.
2. Write a pipeline script to:
   - Clone the Git repository.
   - Build and tag the Docker image.
   - Push the image to Docker Hub.
   - Deploy the application using Kubernetes.
3. Ensure Jenkins has required plugins installed, such as:
   - Git
   - Docker Pipeline
   - Kubernetes CLI

Step 7: Deploy the Application on Kubernetes:

1. Ensure that Minikube is started, and kubectl is pointing to the correct context (minikube).
2. Create a **"k8s"** directory in the project root for Kubernetes configuration files.
3. Create a **"deployment.yaml"** file with the following specifications:
   - Use **node-js-sample** image from Docker Hub.
   - Configure **2 replicas** for the application.
   - Define a **NodePort service** to expose the application on **port 30007**.
4. Apply the deployment to Kubernetes:
   kubectl apply -f k8s/deployment.yaml

Step 8: Test the Kubernetes Deployment:

1. Verify the status of the deployment and service:
   kubectl get pods

kubectl get svc

2. Use the **"minikube ip"** command to find the cluster IP.
3. Access the application in your browser: **http://<minikube-ip>:30007**

Step 9: Add a Persistent Volume and Persistent Volume Claim

1. Create a Persistent Volume (PV): Define a Persistent Volume resource with the following specifications:
   - Name: node-js-pv
   - Storage Capacity: 1Gi
   - Access Mode: ReadWriteOnce
   - Host Path: /data/node-js (path on the node where data is stored)

2. Create a Persistent Volume Claim (PVC) with the following specifications:

   - Name: node-js-pvc
   - Access Mode: ReadWriteOnce
   - Requested Storage: 1Gi
   - Save the definitions in a YAML file named **"persistent-volume.yaml"** in "node-js-sample" folder:

3. Use the appropriate kubectl command to apply the YAML and create the PV and PVC in your cluster.

4. Modify your **deployment.yaml** file to:

   - Mount the PVC (node-js-pvc) to a directory inside the container (/usr/src/app/data).
   - Ensure the volume and volume mount sections are added to the container configuration.

5. Apply the updated deployment to the application to reflect the changes.

6. Confirm the PVC is bound to the PV using "kubectl get pvc".

Step 10: Create a Network Policy to Allow Traffic from Specific Pods

1. Create a Network Policy resource to control incoming traffic to your application. Target pods with the label **app: node-js**. Restrict incoming traffic to only allow traffic from pods with the label **access: allowed**.

2. Create a YAML filewith name **"network-policy.yaml"** in the "node-js-sample" folder for the network policy definition.
3. Network Policy Specifications: Include the following key configurations:

- podSelector: Targets pods with the label **app: node-js**.
- policyTypes: Defines this as an **Ingress** rule for controlling incoming traffic.
- from: Specifies the source pods that are allowed to send traffic, identified by the label **access: allowed**.

4. Use the appropriate kubectl command to apply the network-policy.yaml and enforce the policy.
5. Ensure the policy is created by listing all network policies in the namespace using kubectl.
6. Label one of the running pods to validate the network policy functionality with the label **access: allowed** to communicate with the node-js pods.
7. Confirm that other unlabeled pod cannot establish a connection.

Testcases

1. **Check Jenkins Status:** Verify the command to check the Jenkins service status.
2. **Validate Dockerfile:** Ensure the Dockerfile exists and is correctly configured.
3. **Validate Docker Image:** Verify the Docker image is built successfully.
4. **Check Docker Image Tagging:** Ensure the Docker image is tagged correctly.
5. **Validate Docker Image Push Command:** Verify the command to push the Docker image to a repository.
6. **Check Pipeline Project Creation:** Ensure the Jenkins pipeline project is created successfully.
7. **Validate Kubernetes Deployment File:** Verify the Kubernetes deployment file is present and correctly configured.
8. **Check Minikube Status Command:** Verify the command to check the status of Minikube.
9. **Validate Pods Running Status:** Ensure the deployed pods are running successfully.
10. **Check Kubernetes Service Status Command:** Verify the command to check the status of Kubernetes services.
11. **Validate Kubernetes Apply Command:** Verify the Kubernetes resources are applied successfully using the kubectl apply command.
12. **Check Application Access:** Ensure the application is accessible through the configured service.
13. **Validate Persistent Volume File:** Verify the presence and correctness of the Persistent Volume (PV) configuration file.
14. **Check PVC Binding to PV:** Ensure the Persistent Volume Claim (PVC) is bound to the Persistent Volume (PV).
15. **Validate Network Policy Creation:** Verify that the Kubernetes network policy is created successfully.
16. **Check Pod Labeling:** Ensure at least one pod is labeled correctly as per the configuration.