Establishing a CI/CD Pipeline for Automated Deployments

PHASE 2- SOLUTION ARCHITECTURE

College Name: Smt Kamala And Sri Venkappa M. Agadi College of Engineering & Technology Lakshmeshwar

Group Members:

- **Name:** Shweta Devati

  CAN ID : CAN_33893022

- **Name:** Prajwal Basavaraj Malagi

  CAN ID : CAN_33889738

- **Name:** Shruti B Rampur

  CAN ID : CAN_33891585

- **Name:** Kashamma Fakkirayya Hirematha

  CAN ID : CAN_33899837

SOLUTION ARCHITECTURE

To streamline the deployment process for the Dockerized e-commerce application, we will set up version control, automate code commits, and establish a CI/CD pipeline. The solution architecture will leverage a well-defined directory structure, and tools like Jenkins, GitHub Actions, and IBM Cloud Kubernetes Service.

The architecture focuses on creating a modular directory structure for the application, version control integration with GitHub, and the setup of a CI/CD pipeline. It includes the following components:

1. **Create the main project folder:**

   mkdir ecommerce-app

   cd ecommerce-app

2. **Create the public folder**

   mkdir public

3. **Create the css and js subfolders under public:**

mkdir public\css

mkdir public\js

4. **Create the necessary files in the public folder:**

echo. > public\css\style.css

echo. >public\js\app.js

echo. >public \index.html

5. **Create the server folder and its subfolders:**

mkdir server

mkdir server\controllers

mkdir server\models

mkdir server\routes

6. **Create the necessary files in the server folder:**

echo.>server\controllers\productControl.js

echo. > server\models\productModel.js

echo. > server\routes\productRoutes.js

echo. > server\server.js

7. **Create package.json and README.md in the root directory:**

echo. > package.json

echo. > README.md

After executing the above commands, your directory structure should look as follows:

ecommerce-app/

├── public/

│ ├── css/

│ │ └── style.css

│ ├── js/

│ │ └── app.js

│ └── index.html

├── server/

│ ├── controllers/

│ │ └── productController.j

│ ├── models/

│ │ └── productModel.js

│ ├── routes/

│ │ └── productRoutes.js

│ └── server.js

├── package.json

└── README.md

VERSION CONTROL SETUP

To ensure that the development team is working collaboratively and tracking changes efficiently, we will set up a **GitHub repository** for version control.

1. **Initialize Git in the project:**

   git init

2. **Create a .gitignore file to avoid committing unnecessary files:**
   Echo node_module/>.gitignor
   Echo .evn>.gitignor

3. **Add files to Git:**

   git add .

4. **Commit the initial codebase:**

   git commit -m "Initial commit of ecommerce-app structure"

5. **Create a GitHub repository** (via GitHub's web interface).

6. **Push the local repository to GitHub:**

   git remote add origin<repository_url>

   git push -u origin master

CI/CD PIPELINE DESIGN AND IMPLEMENTATION

To automate the build, test, and deployment processes, we will design a CI/CD pipeline using **Jenkins**.

1. **Jenkins Setup**

   - Install Jenkins on a server (either local or cloud-based) and configure it with necessary plugins like Docker, Git, and Kubernetes CLI.
   - Set up a Jenkins job that triggers on code changes pushed to the GitHub repository.

2. **Jenkins Pipeline Creation**

Using Jenkins, the CI/CD pipeline automates the following steps:

- **Checkout**: Pull the latest code from GitHub.
- **Build**: Create Docker images for the application.
- **Test**: Run unit and integration tests.
- **Push to Registry**: Push Docker images to IBM Cloud Container Registry.
- **Deploy**: Use Kubernetes commands to deploy the application to IBM Cloud Kubernetes Service.

3. **Jenkinsfile Example**:

```
import os

def build_pipeline():
   pipeline_script = """
pipeline {
   agent any
   environment {
      DOCKER_IMAGE = 'my-app'
      REGISTRY_URL = '<IBM_Container_Registry_URL>'
      CLUSTER_NAME = '<CLUSTER_NAME>'
```

```
  }
  stages {
    stage('Checkout') {
      steps {
        git 'https://github.com/<username>/ecommerce-app.git'
      }
    }
    stage('Build Docker Image') {
      steps {
        sh 'docker build -t $REGISTRY_URL/$DOCKER_IMAGE .'
      }
    }
    stage('Push Docker Image to IBM Cloud Container Registry') {
      steps {
        sh 'docker push $REGISTRY_URL/$DOCKER_IMAGE'
      }
    }
    stage('Deploy to Kubernetes') {
      steps {
        sh '''
        ibmcloud login --apikey <API_KEY> -r <REGION> -g <RESOURCE_GROUP>
        ibmcloud ks cluster config --cluster $CLUSTER_NAME
        kubectl apply -f k8s/deployment.yaml
        '''
      }
    }
  }
  post {
    success {
      echo 'Pipeline executed successfully.'
    }
    failure {
      echo 'Pipeline failed. Please check the logs.'
    }
  }
}
```

FUTURE PLAN:

1. **Container Image Management**:
   - Use IBM Cloud Container Registry for secure storage and management of Docker images.
2. **Enhanced Security**:
   - Implement OpenSSL for encrypting and signing Docker images.
   - Conduct vulnerability scanning to ensure secure deployments.
3. **Scaling and Resilience**:
   - Simulate production environments using Minikube for local testing.
   - Deploy to Kubernetes clusters for scalable, resilient application hosting.
4. **CI/CD Integration**:
   - Extend CI/CD automation using tools like IBM Cloud Continuous Delivery or GitHub Actions.
5. **Kubernetes Cluster Setup**:

   - Use Minikube for local testing and IBM Kubernetes Service for production.