Deploying a Spring-boot Application on AWS EKS using Jenkins CICD

- Example Project:
https://github.com/hosniah/springboot-app-for-aks

- Creating and Managing EKS Clusters

- Install Jenkins on an AWS Linux EC2:

  sudo yum update –y
  sudo wget -O /etc/yum.repos.d/jenkins.repo
  https://pkg.jenkins.io/redhat-stable/jenkins.repo
  sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io.key
  sudo yum upgrade
  sudo amazon-linux-extras install java-openjdk11 -y
  sudo yum install jenkins -y --nogpgcheck
  sudo systemctl enable jenkins
  sudo systemctl start jenkins
  sudo systemctl status jenkins

- Connect to http://<instance_public_ip>:8080 from your browser. You will be able to access Jenkins through Management Interface.
- Now on the left-hand side, go to Manage Jenkins and then select Manage Plugins
- Go to the Available tab and then type **Amazon EC2** plugin at the top right.

- Select Manage Jenkins and then select Manage Nodes and Clouds
- Select Configure Cloud and then Add a new cloud and select Amazon EC2
- In the fields that appear on the window, Give some name to Amazon EC2, Click Add under Amazon EC2 Credentials
- Select AWS Credentials as the Kind from the Jenkins Credentials Provider.
- Enter the IAM User programmatic access keys with EC2 instance launch permissions and click Add.
- Select region from the drop-down and Add EC2 Key Pair's Private Key,
- Select an SSH Username with Private Key as the Kind and ec2-user as the Username from the Jenkins Credentials Provider.
- Enter your Private Key directly and click on Add
- Click on Test connection and make sure that it states "Success."

- **Install Docker on Amazon Linux Machine**

sudo yum update -y
sudo yum install docker -y
sudo systemctl start docker
sudo docker run hello-world
sudo systemctl enable docker
docker --version
sudo usermod -a -G docker $(whoami)
newgrp docker

- **Install required Plugins in Jenkins**

  Amazon EC2 plugin
  Amazon ECR plugin
  Docker plugin
  Docker Pipeline
  CloudBees Docker Build and Publish plugin
  Kubernetes CLI Plugin
  Pipeline: AWS Steps

- **Integrate Docker with Jenkins**
- Add Jenkins user to the Docker group:
  sudo usermod -a -G docker jenkins
  sudo systemctl restart jenkins
  sudo systemctl daemon-reload
  sudo service docker stop
  sudo service docker start

- **Create a repository in ECR**

- Log in to the AWS Management Console and navigate to the Amazon ECR service.
- Click on the “Create repository” button.
- Enter a name for your repository. This name must be unique within your AWS account.
- (Optional) Add a description for your repository.
- Click on the “Create repository” button.
- You will now see your newly created repository in the repository list.
- Select the newly created repo and then choose to view push commands.
- Use those commands to authenticate and push an image to your repository while writing Jenkinsfile.

- **Add Maven to Jenkins**

```
        tools {
                maven 'Maven3'
        }
```

- **Write Jenkinsfile**

- Write a Jenkinsfile to define the steps in a pipeline for deploying a spring-boot application to an EKS cluster using Jenkins.

```
pipeline {
    tools {
        maven 'Maven3'
    }
    agent any
    stages {
        stage('Checkout') {
            steps {
                checkout([$class: 'GitSCM', branches: [[name: '*/main']], extensions: [],
userRemoteConfigs: [[url: '<GIT_REPO_URL>']]])
            }
        }
        stage('Build Jar') {
            steps {
                sh 'mvn clean package'
            }
        }
        stage('Docker Image Build') {
            steps {
                sh 'docker build -t <IMAGE_NAME> .'
            }
        }
        stage('Push Docker Image to ECR') {
            steps {
                withAWS(credentials: '<AWS_CREDENTIALS_ID>', region: '<AWS_REGION>') {
                    sh 'aws ecr get-login-password --region <AWS_REGION> | docker login
--username AWS --password-stdin <ECR_REGISTRY_ID>'
                    sh 'docker tag <IMAGE_NAME>:latest
<ECR_REGISTRY_ID>/<IMAGE_NAME>:latest'
                    sh 'docker push <ECR_REGISTRY_ID>/<IMAGE_NAME>:latest'
                }
            }
        }
        stage('Integrate Jenkins with EKS Cluster and Deploy App') {
            steps {
```

```
            withAWS(credentials: '<AWS_CREDENTIALS_ID>', region: '<AWS_REGION>') {
              script {
                sh ('aws eks update-kubeconfig --name <EKS_CLUSTER_NAME> --region
<AWS_REGION>')
                sh "kubectl apply -f <K8S_DEPLOY_FILE>.yaml"
            }
            }
      }
   }
   }
}
```

- Interact with a cluster from terminal

- Retrieve the status of an Amazon Elastic Container Service for Kubernetes (EKS) cluster
  aws eks describe-cluster --region <region-name> --name <cluster-name> --query cluster.status

- Update the kubeconfig file
  aws eks --region <region-name> update-kubeconfig --name <cluster-name>

- Retrieve data from the cluster
  kubectl get nodes
  kubectl get pods
  kubectl get services
  kubectl get deployments

- Expose the service
- Get the external IP
- Allow the required ports

- Install EKS cluster:

- In AWS CloudShell:

```
curl --silent --location
"https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname
-s)_amd64.tar.gz" | tar xz -C /tmp
sudo mv /tmp/eksctl /usr/bin
eksctl version
eksctl create cluster --name dev --version 1.24 --region us-east-1 --nodegroup-name
standard-workers --node-type t3.micro --nodes 2 --nodes-min 1 --nodes-max 3 --managed
aws eks update-kubeconfig --name dev --region us-east-1
```

- Setting Up Kubernetes Namespace & Service Account

- Step 1: Create a namespace called devops-tools

```
kubectl create namespace devops-tools
```

- Step 2: Save the following manifest as service-account.yaml.
It contains the role and role-binding for the service account with all the permission to manage pods in the devops-tools namespace.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins-admin
  namespace: devops-tools
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: jenkins
  namespace: devops-tools
  labels:
    "app.kubernetes.io/name": 'jenkins'
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
```

```
  resources: ["pods/log"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: jenkins-role-binding
  namespace: devops-tools
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: jenkins
subjects:
- kind: ServiceAccount
  name: jenkins-admin
  namespace: devops-tools
```

- Create the service account.

kubectl apply -f service-account.yaml

- Create a secret with the type service-account-token

Create a secret with the type service-account-token and pass the service-account name in the annotation section as shown below.

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  namespace: devops-tools
  annotations:
    kubernetes.io/service-account.name: "jenkins-admin"
type: kubernetes.io/service-account-token
data:
  # You can include additional key value pairs as you do with Opaque Secrets
  extra: YmFyCg==
```

- **Jenkins server running Outside the Kubernetes cluster**

- Kubernetes URL:

- Kubernetes Server Certificate key:

If you have a Kubernetes Cluster CA certificate, you can add it for secure connectivity. You can get the certificate from the pod location /var/run/secrets/kubernetes.io/serviceaccount/ca.crt .If you do not have the certificate, you can enable the “disable https certificate check” option.

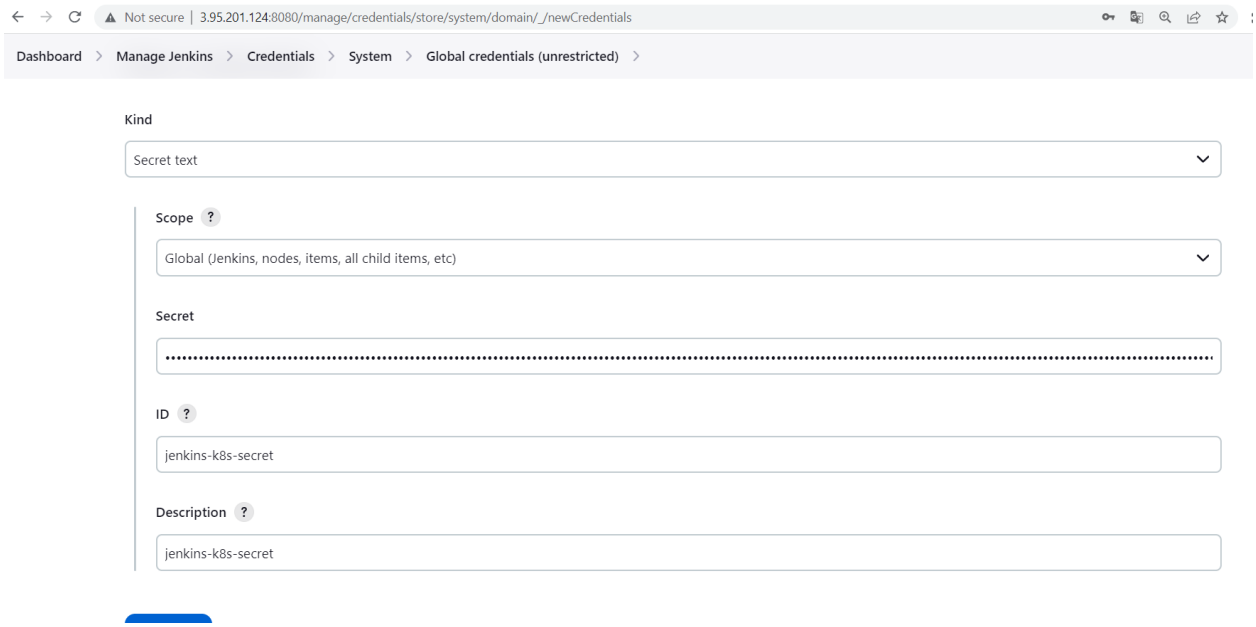
You will need to decode it (it is base64encoded in the aws EKS GUI):

[cloudshell-user@ip-10-10-31-13 ~]$ echo
'LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUMvakNDQWVhZ0F3SUJBZ0lCQURBT
kJna3Foa2lHOXcwQkFRc0ZBREFWTVJNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVn
pNQjRYRFRJek1EVXhOekU0TWpnd09Gb1hEVE16TURVeE5ERTRNamd3T0Zvd0ZURVRNQk
VHQTFVRQpBeE1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBR
GdnRVBBRENDQVFvQ2dnRUJBTHZPCi9aeHZVSkEwQ0pkTUZ0dUJtTlpMTUY3LzhIVjlWUkl
wM1BqMk81Y2NMN04wVUloZkxwVFh4RWlTRS9vZEVFN1QKaEVxd25YL0RTSmV5VElYbUx
qRjlTaDVYY3FhMmlnc3RiV0c0TUQvbXdrUGlwY0xBbFh0czFrNmdqcERuV3pORAp0cGZ3NzM
ydVpXYWpITlcxY1FETU5oU1U4TmpVYTdXK3hXMHN5UERYajZrMW16c0tSWml0SUhvbHp1Z
DFTRWxSCnJVWXI4d042OXhDYUNKdzl0L3RNSmVmU1VrSTI5RU1ZUEpneVkzR0VIc0YrQW
ZTNVBHNXVFbVZMVnU2dERSSkcKR1RsTUNqeUlaMG1zdmJQZ1VTQjJDblJsRURjSFdEaC8
wUlNsTFNseDhMeGU2cURnaUZ0TGFuRkhFNEkySW1GVwpXNEtPWThCY25UTWxVNTk4bn
FNQ0F3RUFBYU5aTUZjd0RnWURWUjBQQVFIL0JBUURBZ0trTUE4R0ExVWRFd0VCCi93UU
ZNQU1CQWY4d0hRWURWUjBPQkJZRUZNSDJmNTFuT3pJS2c5Q3ByVHU1THV0Y2kwZ1Z
NQlVHQTFVZEVRUU8KTUF5Q0NtdDFZbVZ5Ym1WMFpYTXdEUVlKS29aSWh2Y05BUUVMQ
lFBRGdnRUJBRmFrQ1JRME5BaWxRbkhQZENtYwpOc2VIbTlkeFRVNWlqVi84OGh1RUtuNEV
nNW52SldCdE9rNmt0eWxzSzRHWUdKYTJsWlcwMmpEeEl5OHUxV3dLCnhub3NpS1BNTytZ
SERmRGFiRWpFNFRGc20zWXN1VW90dkE2bTdnV1FMRnJSRHpCMGdwNGxrV3NaRUloNn
c0eTMKbGprSTJMYk5KcmZtckhEZG9mUktUS0dEMjVwcU5CUEJJUVNGbWw4c2x1dmp1dUp
DbllwbDZyVVZqcERXbjhaUgpuV1lSa01TTFIvc2x0NXlTZU9pNnkzMDI5U0UvTkRnZGlWcTFBT
zFxTEF1eWhYd3gxSlVrM0kvYWFLem54dTZCCnE1bGRramgyUEQ1N1UwaHRWTlU0MlZGN
CsvNVNoNnI1aEF2RjdxZ2dWRjF4R3VBdS9jdlQwdW03eGlkaHZwQ0QKRUxrPQotLS0tLUVO
RCBDRVJUSUZJQ0FURS0tLS0tCg==' | base64 -d
-----BEGIN CERTIFICATE-----
MIIC/jCCAeagAwIBAgIBADANBgkqhkiG9w0BAQsFADAVMRMwEQYDVQQDEwprdWJl
cm5ldGVzMB4XDTIzMDUxNzE4MjgwOFoXDTMzMDUxNDE4MjgwOFowFTETMBEGA1UE
AxMKa3ViZXJuZXRlczCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALvO
/ZxvUJA0CJdMFtuBmNZLMF7/8HV9VRIp3Pj2O5ccL7N0UIhfLpTXxEiSE/odEE7T
hEqwnX/DSJeyTIXmLjF9Sh5Xcqa2igstbWG4MD/mwkPipcLAlXts1k6gjpDnWzND
tpfw732uZWajHNW1cQDMNhSU8NjUa7W+xW0syPDXj6k1mzsKRZitIHolzud1SElR
rUYr8wN69xCaCJw9t/tMJefSUkI29EMYPJgyY3GEHsF+AfS5PG5uEmVLVu6tDRJG
GTlMCjyIZ0msvbPgUSB2CnRlEDcHWDh/0RSlLSlx8Lxe6qDgiFtLanFHE4I2ImFW
W4KOY8BcnTMlU598nqMCAwEAAaNZMFcwDgYDVR0PAQH/BAQDAgKkMA8GA1UdEwEB
/wQFMAMBAf8wHQYDVR0OBBYEFMH2f51nOzIKg9CprTu5Lutci0gVMBUGA1UdEQQO
MAyCCmt1YmVybmV0ZXMwDQYJKoZIhvcNAQELBQADggEBAFakCRQ0NAilQnHPdCmc
NseHm9dxTU5ijV/88huEKn4Eg5nvJWBtOk6ktylsK4GYGJa2lZW02jDxIy8u1WwK

xnosiKPMO+YHDfDabEjE4TFsm3YsuUotvA6m7gWQLFrRDzB0gp4lkWsZEIh6w4y3
ljkI2LbNJrfmrHDdofRKTKGD25pqNBPBIQSFml8sluvjuuJCnYpl6rUVjpDWn8ZR
nWYRkMSLR/slt5ySeOi6y3029SE/NDgdiVq1AO1qLAuyhXwx1JUk3I/aaKznxu6B
q5ldkjh2PD57U0htVNU42VF4+/5Sh6r5hAvF7qggVF1xGuAu/cvT0um7xidhvpCD
ELk=
-----END CERTIFICATE-----

- **Credentials:**

For Jenkins to communicate with the Kubernetes cluster,
we need a service account token with permission to deploy pods in the target ns (let's say devops-tools namespace).



- get the base64 encoded service account token and then decode it:

kubectl get secrets $SECRET_NAME -o=jsonpath='{.items..data.token}' -n devops-tools | base64 -d

- Go under credentials and create a credential type “Secret text“.

Enter the service account token in the secret box.

After filling in all the details, you can test the connection to validate the Kubernetes cluster connectivity.

- Also, add the POD label that can be used for grouping the containers if required in terms of billing or custom build dashboards.

- Create POD and Container Template

The label kubeagent will be used in the job as an identifier to pick this pod as the build agent. Next, we need to add a container template with the Docker image details.

- **Jenkinsfile With Pod Template**

When it comes to actual project pipelines, it is better to have the POD templates in the Jenkinsfile

Here is what you should know about the POD template.

1- By default, a JNLP container image is used by the plugin to connect to the Jenkins server. You can override with a custom JNLP image provided you give the name jnlp in the container template.
2- You can have multiple container templates in a single pod template. Then, each container can be used in different pipeline stages.
3- POD_LABEL will assign a random build label to the pod when the build is triggered. You cannot give any other names other than POD_LABEL

Here is an example Jenkinsfile with a POD template.

```
podTemplate {
    node(POD_LABEL) {
        stage('Run shell') {
            sh 'echo hello world'
        }
    }
}
```

here, instead of jenkins/inbound-agent:latest, you will have your own image.

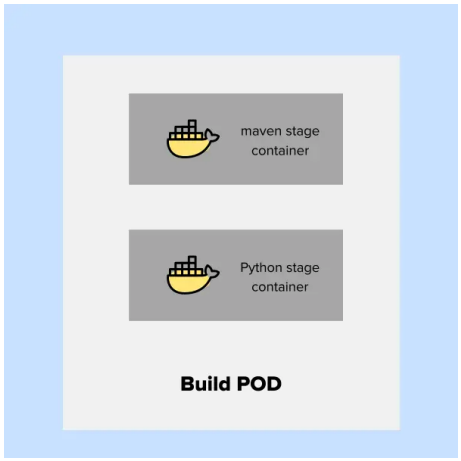
```
podTemplate(containers: [
    containerTemplate(
        name: 'jnlp',
        image: 'jenkins/inbound-agent:latest'
        )
  ]) {

    node(POD_LABEL) {
        stage('Get a Maven project') {
            container('jnlp') {
                stage('Shell Execution') {
                    sh '''
                    echo "Hello! I am executing shell"
                    '''
                }
            }
        }

    }
}
```

- **Multi Container Pod Template**

You can use multiple container templates in a single POD template.




```
podTemplate(containers: [
    containerTemplate(
        name: 'maven',
        image: 'maven:3.8.1-jdk-8',
        command: 'sleep',
        args: '30d'
        ),
    containerTemplate(
        name: 'python',
        image: 'python:latest',
        command: 'sleep',
        args: '30d')
  ]) {

    node(POD_LABEL) {
        stage('Get a Maven project') {
          git branch: 'main', url: 'https://github.com/spring-projects/spring-petclinic.git'
           container('maven') {
               stage('Build a Maven project') {
                   sh '''
                   echo "maven build"
                   '''
               }
           }
        }

        stage('Get a Python Project') {
            git url: 'https://github.com/hashicorp/terraform.git', branch: 'main'
            container('python') {
                stage('Build a Go project') {
                    sh '''
```

```
                echo "Go Build"
                '''
            }
        }
    }

  }
}
```

- **Using Shared Persistent Volumes With Jenkins Docker Agent Pods**

To speed up the build process, it is better to attach a shared persistent volume to the build container.
The .m2 cache is not possible in Docker agent-based builds as it gets destroyed after the build.

We can create a persistent volume for the maven cache and attach it to the agent pod via the container template to solve this issue.

To demonstrate this, first, let's create a PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: maven-repo-storage
  namespace: devops-tools
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi
```

Here is an example Jenkinsfile with POD template that uses the maven-repo-storage persistent volume.

```
podTemplate(containers: [
  containerTemplate(
      name: 'maven',
      image: 'maven:latest',
      command: 'sleep',
      args: '99d'
      )
```

```
  ],

  volumes: [
  persistentVolumeClaim(
      mountPath: '/root/.m2/repository',
      claimName: 'maven-repo-storage',
      readOnly: false
      )
  ])

{
  node(POD_LABEL) {
    stage('Build Petclinic Java App') {
      git url: 'https://github.com/spring-projects/spring-petclinic.git', branch: 'main'
      container('maven') {
        sh 'mvn -B -ntp clean package -DskipTests'
      }
    }
  }
}
```