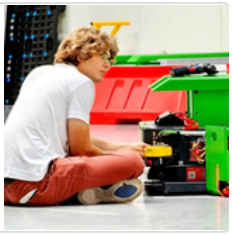
Kubernetes EKS: Deployment & containerization of Node.js, mongoDB, React webapp with Secure Jenkins CI/CD, GitOps principles, Terraform good practices, Grafana&Prometheus observability Stack

Check GitHub for helpful DevOps tools:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



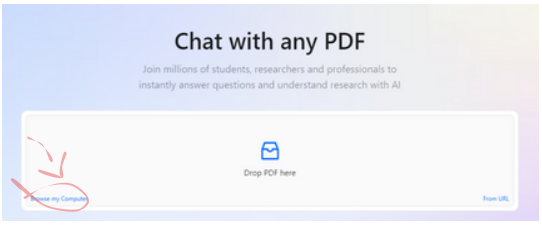
Ask Personal AI Document assistant to learn interactively (FASTER)!

1. Download PDF
2. Go to website
3. Browse file
4. Chat with Document

1 https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesEKSNodejs.pdf

Click there to go to ChatPdf website 2

3



Ask questions about document! 4

Complety new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

https://www.hackthebox.com/

What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes EKS: Jenkins server setup

1) Jenkins server setup steps, local environemnt access setup

First we will create an AWS IAM user with the necessary permissions for deployment and management. Then, Terraform and AWS CLI will be used to provision a Jenkins EC2 instance. Once the server is up we will install&configure, Jenkins, Docker, SonarQube, Terraform, Kubectl, AWS CLI, and Trivy to ensure a fully functional CI/CD environment.

To create an access point for your AWS account, go to AWS IAM, navigate to **Users**, and select **Create User.** Enter a name for your user, then under **Permissions**, choose **"Attach policies directly"** and find the **"AdministratorAccess"** policy. After selecting it, click **Next** and then **Create User**. Once the user is created, go to **Security Credentials**, select **Create Access Key**, and choose **Command Line Interface (CLI)**. Name your key, create the access key, and make sure to save the Access Key ID and Secret Access Key securely. Your access point is now set up.

Now install teraform and aws CLI on your linux machine with your Aws access credentials, there are plenty tutorials out there.

2) Provision EC2 with Jenkins and other tools

According to industry standards, we will provision an EC2 instance with Jenkins and all other essential tools installed. You will need to create your own repository—do not clone. Copy, or fork this repository:

https://github.com/MichaelRobotics/Kubernetes/tree/main/EKSNodejs

The Terraform files are located in Kubernetes/EKSNodejs and contain a basic EC2 deployment. For configuring subnets and the region, refer to the vpc.tf file.

EKSNodejs > vpc.tf

```
resource "aws_vpc" "vpc" {
  cidr_block = "10.0.0.0/16"

  tags = {
    Name = var.vpc-name
  }
}

resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.vpc.id

  tags = {
    Name = var.igw-name
  }
}

resource "aws_subnet" "public-subnet" {
  vpc_id                  = aws_vpc.vpc.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = "us-east-1a"
```

If you change the region, don't forget to update the values in provider.tf.

EKSNodejs > provider.tf

```
provider "aws" {
  region = "us-east-1"
}
```

To modify the EC2 instance size, navigate to ec2.tf. Check the user_data section, which contains the path to a script that runs immediately after provisioning. This script includes installation instructions for essential tools like Jenkins, Sonar, Trivy, and others.

EKSNodejs > ec2.tf

```
resource "aws_instance" "ec2" {
  ami                    = data.aws_ami.ami.image_id
  instance_type          = "t2.2xlarge"
  key_name               = var.key-name
  subnet_id              = aws_subnet.public-subnet.id
  vpc_security_group_ids = [aws_security_group.security-group.id]
  iam_instance_profile   = aws_iam_instance_profile.instance-profile.name
  root_block_device {
    volume_size = 30
  }
  user_data = templatefile("./tools-install.sh", {})

  tags = {
    Name = var.instance-name
  }
}
```

To successfully initialize the project, one key step remains: provide the PEM key used to log into your EC2 instance. If you don’t have one, generate a new key in the same region. Check variables.tfvars to ensure it's correctly set.

EKSNodejs > variables.tfvars

```
vpc-name      = "Jenkins-vpc"
igw-name      = "Jenkins-igw"
subnet-name   = "Jenkins-subnet"
rt-name       = "Jenkins-route-table"
sg-name       = "Jenkins-sg"
instance-name = "Jenkins-server"
key-name      = "<your-key-name>"
iam-role      = "Jenkins-iam-role"
```

Infrastructure as Code (IaC) has been implemented following best practices, including state-locking. To configure this, navigate to backend.tf and update the S3 bucket, DynamoDB table, and the Terraform state (tfstate) file location.

Comment out the contents of backend.tf then run

EKSNodejs > backend.tf

```
terraform {
  backend "s3" {
    bucket         = "<terraform-state-bucket-name>"
    region         = "us-east-1"
    key            = "<terraform.tfstate-file-location>"
    dynamodb_table = "<your-dynamodb-table-name>"
    encrypt        = true
  }
  required_version = ">=0.13.0"
  required_providers {
    aws = {
      version = ">= 2.7.0"
      source  = "hashicorp/aws"
```

```
terraform init
```

Errors will appear, but the tfstate file will be generated. Modify backend.tf with the correct values. Run again.

```
terraform init
```

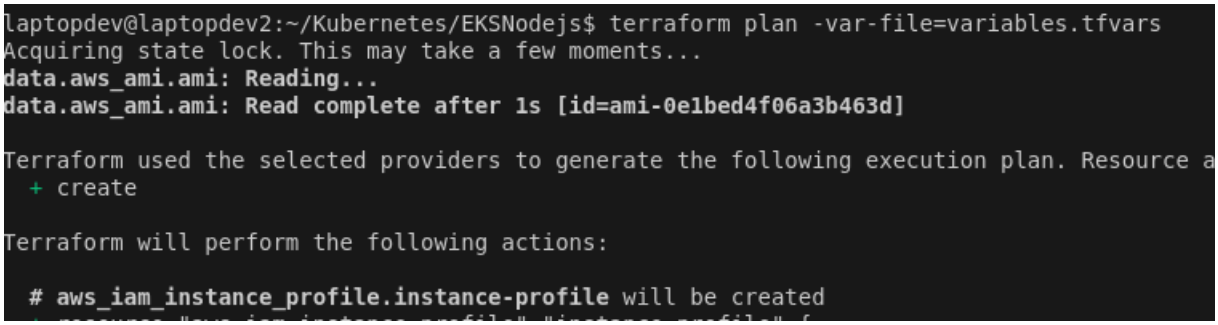
All remaining .tf files define the necessary IAM roles to set up the EC2 instance properly.

To validate and check for any mistakes in your Terraform configuration, run

```
terraform validate
```

If everything is okay, proceed with planning by specifying the path to your variables file:

```
terraform plan -var-file=variables.tfvars
```



Once confirmed, apply the configuration:

```
terraform apply -var-file=variables.tfvars --auto-approve
```

3) Configure jenkins

First, SSH into the EC2 instance where Jenkins is installed.

Go to your AWS account, navigate to EC2 Instances, and choose to connect via SSH.

You should find a command similar to this:

```
ssh -i "EKSNodejs.pem" ubuntu@54.172.216.169
```

Remember give your key 400 permision before connecting

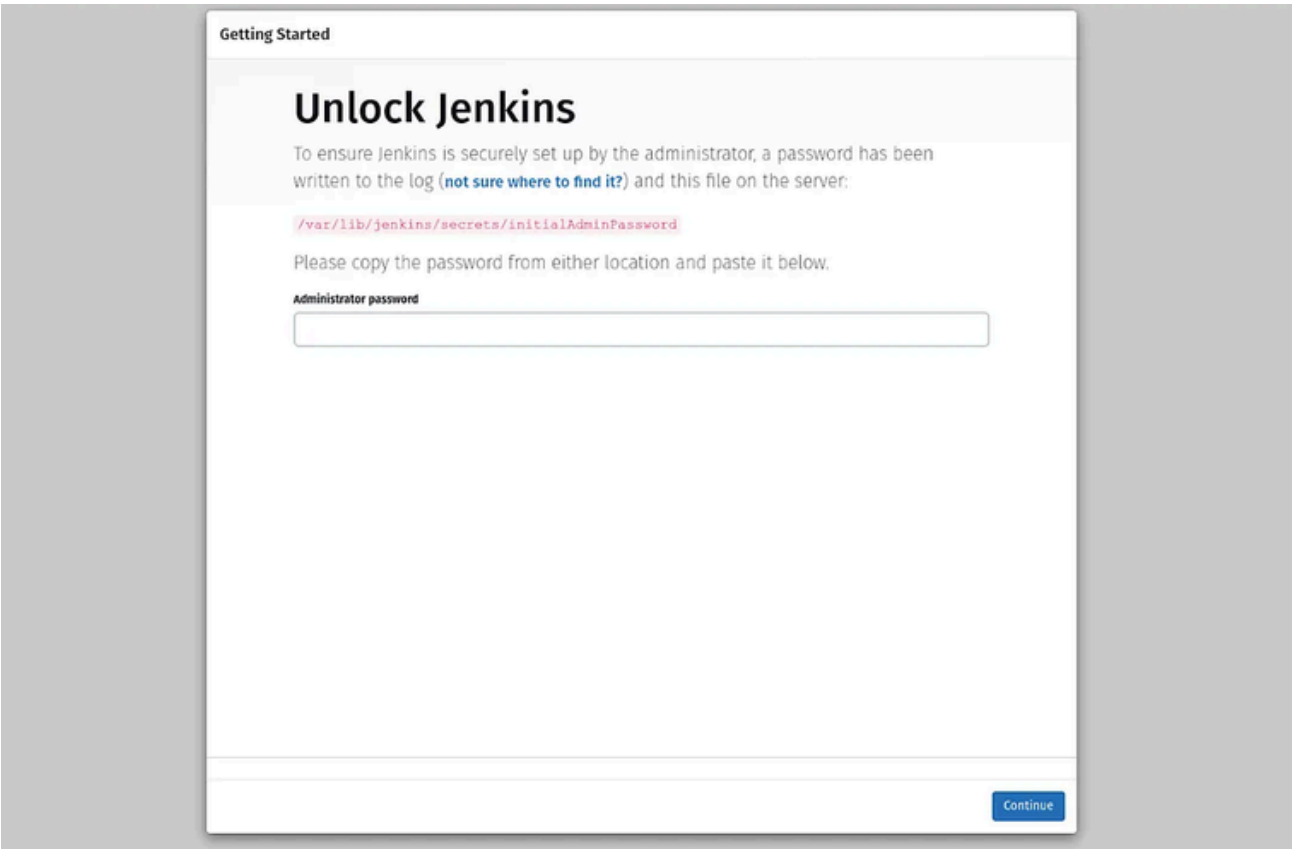
```
chmod 400 EKSNodejs.pem
```

Check whether all tools are installed or not:

```
jenkins --version
docker --version
docker ps
terraform --version
kubectl version
aws --version
trivy --version
eksctl version
```

Now go to aws and get your ec2 public ip. Paste it with jenkins port (8080) into web browser.

Follow standard jenkins installation procedures.



Kubernetes EKS: Setup EKS cluster, ECR, Sonarqube, Install ArgoCD

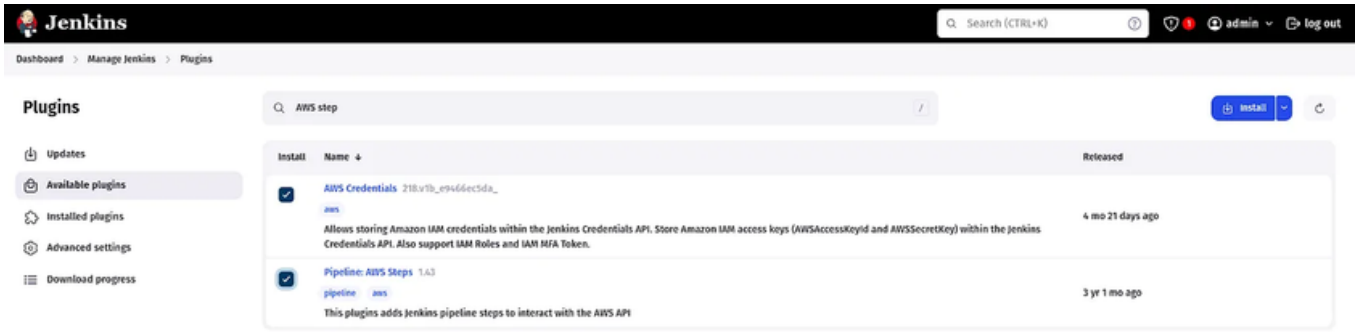
1) Credentials

We will create the EKS cluster from the newly created EC2 instance, but it doesn't have AWS access by default. To provide it with access, run and provide credentials:

```
aws configure
```

Since Jenkins will be using the ECR registry, we need to grant it permission to access AWS as well. To do so: Dashboard-> Manage Jenkins-> Plugins -> available plugins and install

- AWS credentials
- Pipeline AWS steps



Log in into your jenkins again. Now naviage towards Dashboard->Manage Jenkins->Credentials and click global. Then setup key accordingly:

2) EKS Setup

Jenkins and ec2 instance should have acces to our aws now. We are ready to setup EKS. On EC2 instance:

```
eksctl create cluster --name Three-Tier-K8s-EKS-Cluster --region us-east-1 --node-type t2.medium --nodes-min 2 --nodes-max 2
aws eks update-kubeconfig --region us-east-1 --name Three-Tier-K8s-EKS-Cluster
```

Check for nodes to make sure if installation was successful

```
kubectl get nodes
```

To add a load balancer to your setup, the first step is to create an IAM role that will allow the load balancer to interact with your AWS environment.
Download policies json

```
curl -O https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/v2.5.4/docs/install/iam_policy.json
```

create iam

```
aws iam create-policy --policy-name AWSLoadBalancerControllerIAMPolicy --policy-document file://iam_policy.json
```

Create OIDC Provider to integrate Kubernetes workloads with AWS IAM

```
eksctl utils associate-iam-oidc-provider --region=us-east-1 --cluster=Three-Tier-K8s-EKS-Cluster --approve
```

Create a Service Account, replace your account ID with your one

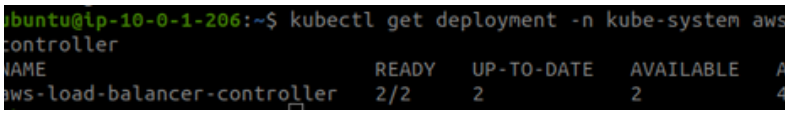
```
eksctl create iamserviceaccount --cluster=Three-Tier-K8s-EKS-Cluster --namespace=kube-system --name=aws-load-balancer-controller --role-name AmazonEKSLoadBalancerControllerRole --attach-policy-arn=arn:aws:iam::<your_account_id>:policy/AWSLoadBalancerControllerIAMPolicy --approve --region=us-east-1
```

Deploy LoadBalancer controller to our k8s

```
sudo snap install helm --classic
helm repo add eks https://aws.github.io/eks-charts
helm repo update eks
helm install aws-load-balancer-controller eks/aws-load-balancer-controller -n kube-system --set clusterName=my-cluster --set serviceAccount.create=false --set serviceAccount.name=aws-load-balancer-controller
```

check if everything is running after couple of minutes:

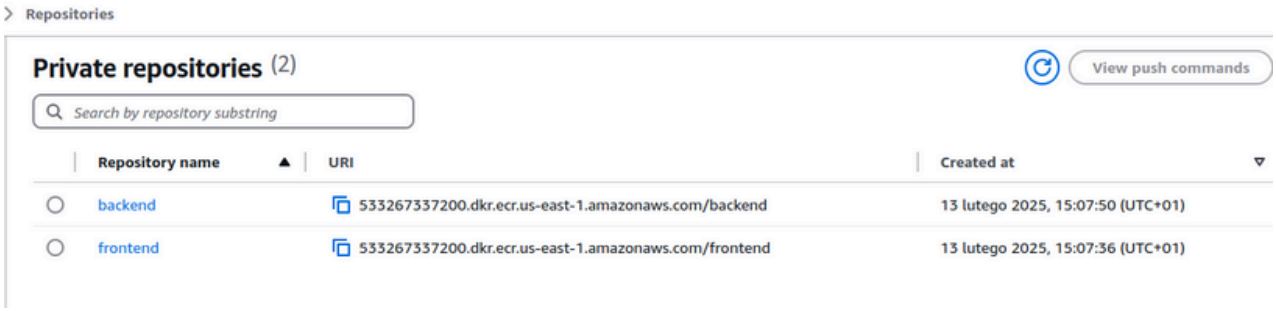
```
kubectl get deployment -n kube-system aws-load-balancer-controller
```



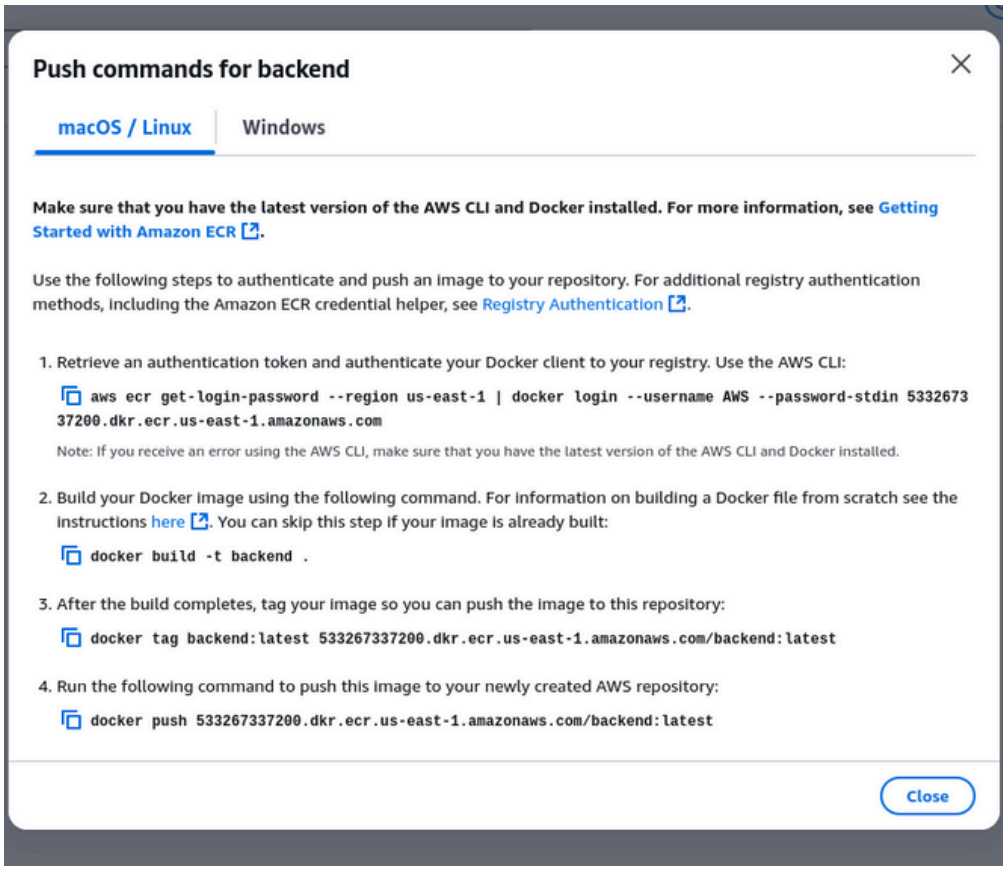
3) ECR Setup

Navigate towards ECR service. Create two repositories

1. Frontend
2. Backend



Then click on one of those, navigate towards “view push commands” and copy script:



Run command on your jenkins server

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 533267337200.dkr.ecr.us-east-1.amazonaws.com
```

create new namespace for our application:

```
kubectl create namespace three-tier
```

create secrets, which will later be used to authenticate to ECR registries (since those are private)

```
kubectl create secret generic ecr-registry-secret \
 --from-file=.dockerconfigjson=${HOME}/.docker/config.json \
 --type=kubernetes.io/dockerconfigjson --namespace three-tier
kubectl get secrets -n three-tier
```

4) ArgoCD insallation

Install Argo in new namespace

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/v2.4.7/manifests/install.yaml
```

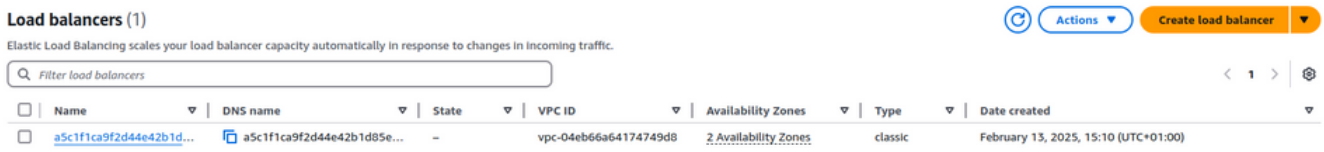
After a moment, verify installation:

```
kubectl get pods -n argocd
```

expose the argoCD server service as LoadBalancer so we will be able to access its UI

```
kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'
```

Navigate toward LoadBalancer created in our aws and save its DNS



Pass DNS into web browser, you will got a pop up of “connection not private”. Navigate towards “Hide advanced” and click url. But before that we need to get password for our argocd.

Install jq so we will be able to process data from kubernetes:

```
sudo apt install jq -y
```

Get argocd server paswword

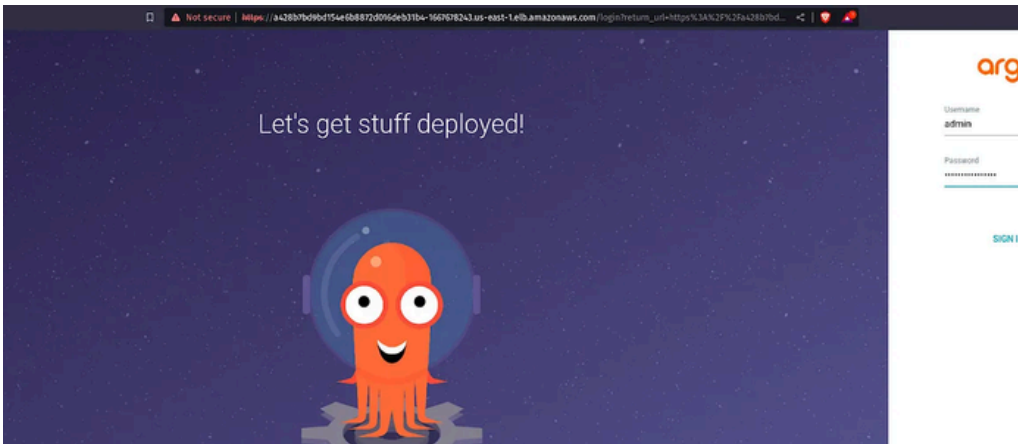
```
export ARGO_PWD=$(kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d)
```

```
echo $ARGO_PWD
```

copy password

```
ubuntu@ip-10-0-1-206:~$ export ARGO_PWD=$(kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d)
ubuntu@ip-10-0-1-206:~$ echo $ARGO_PWD
T13M0p6PSMWBnMA2
ubuntu@ip-10-0-1-206:~$
```

and login

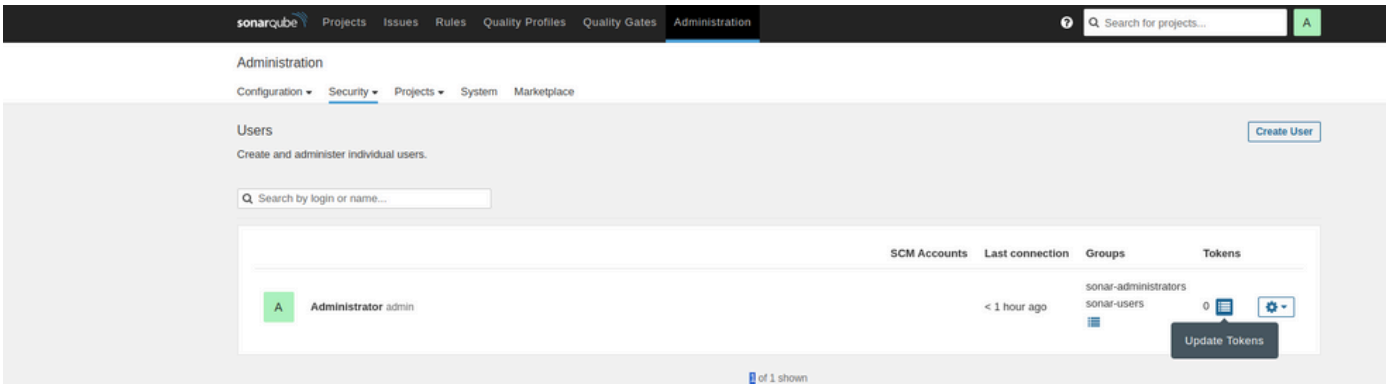


5) Sonarqube insallation

Paste into URL EC2 ip (same as jenkins) but with 9000 port. Username and password are admin

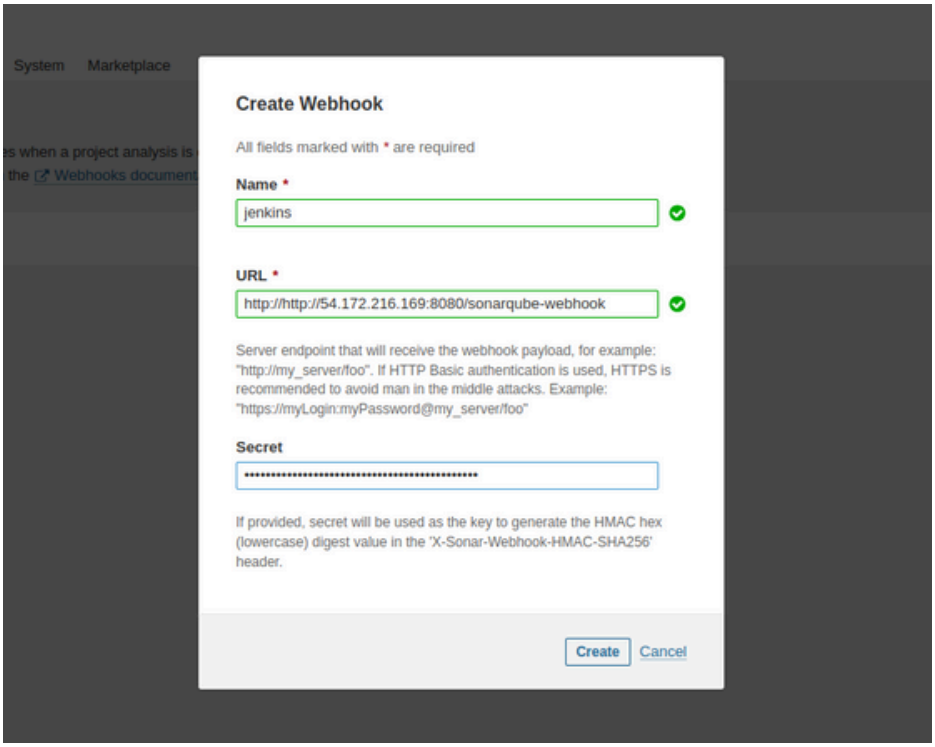
Click Administration->Security->Users->Update tokens->Generate

copy token and keep it somewhere -> Done

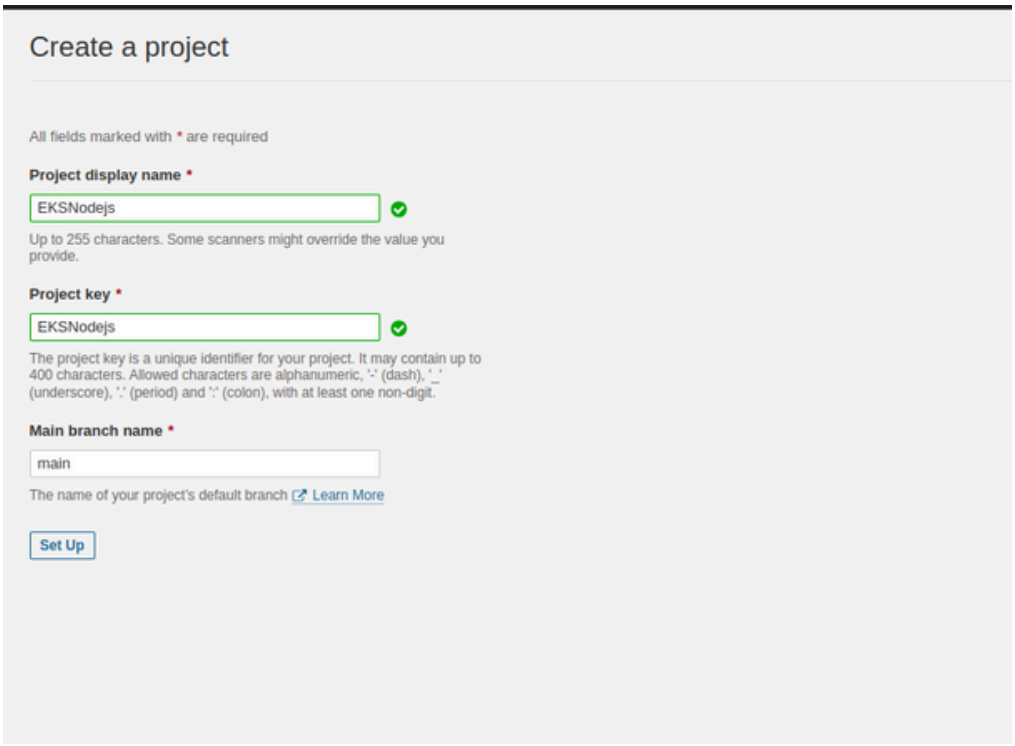


Now lets configure webhook which will make jenkins triger quality checks

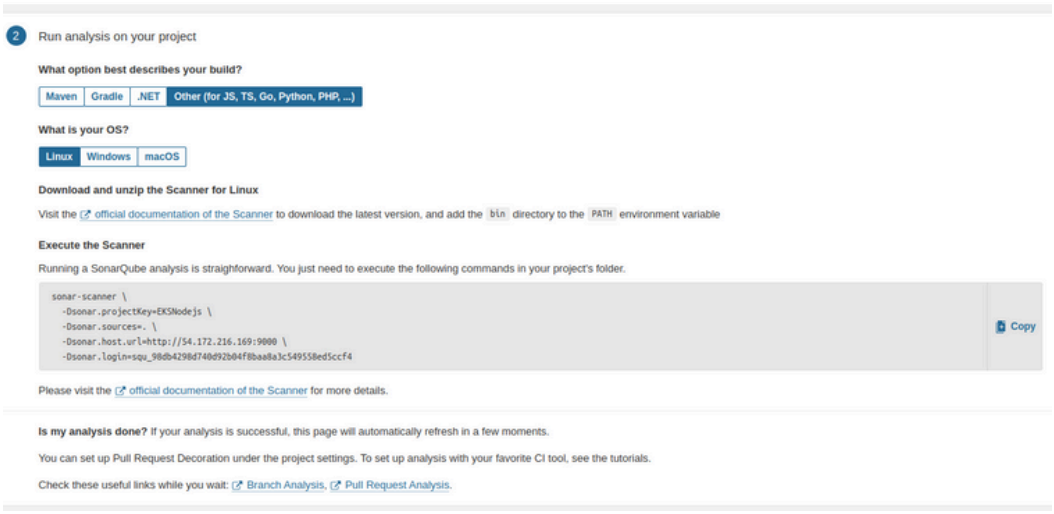
Administration -> Webhooks -> Create



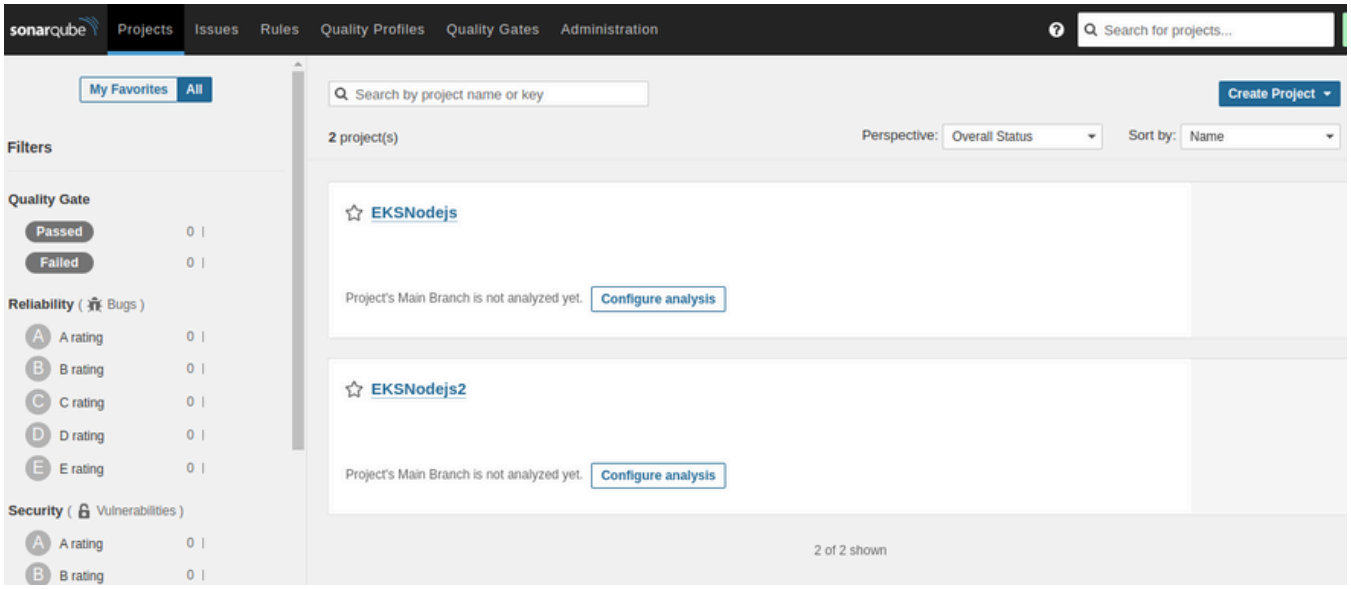
Now create Project for frontend code. Go Projects-> Create new project -> Manually. Name your project:



Click Use existing token-> select other and Linux as OS. Save sonnar-scanner code, we will use it in Jenkins pipeline to perform sonarqube scan for frontend repo



Now repeat same actions for creation of backend project. You should have 2 projects now:

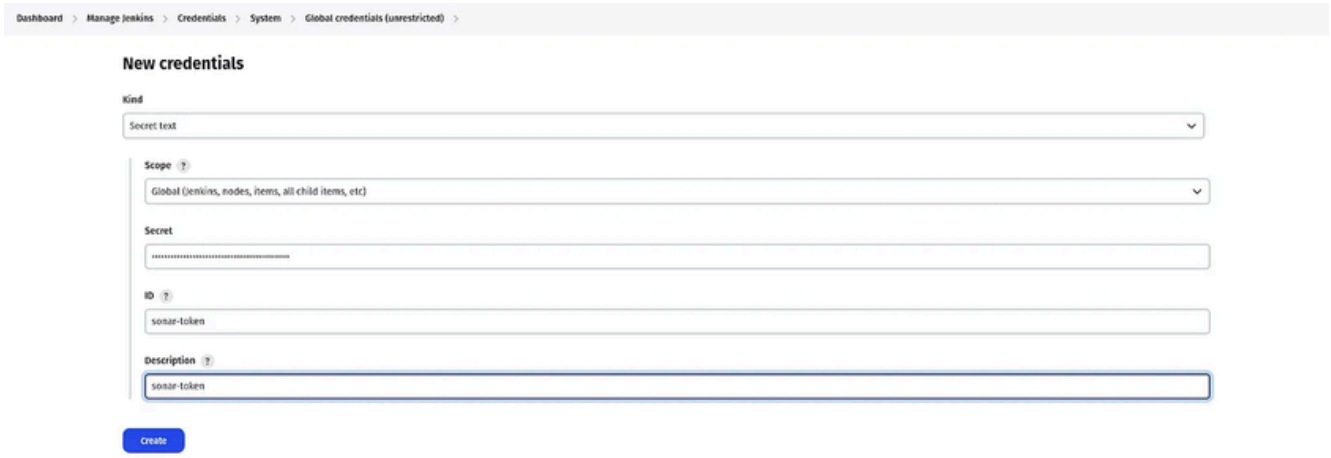


Now its time to setup credentials for jenkins to authenticate into:

- Sonar (use sonar token)
- github(use PAT personal acces token)
- ECR(we need to pass account ID and ecr frontend&backend names)
- NVD(National Vulnerability Database. We need to get from them our API key for vulnarebility checks)

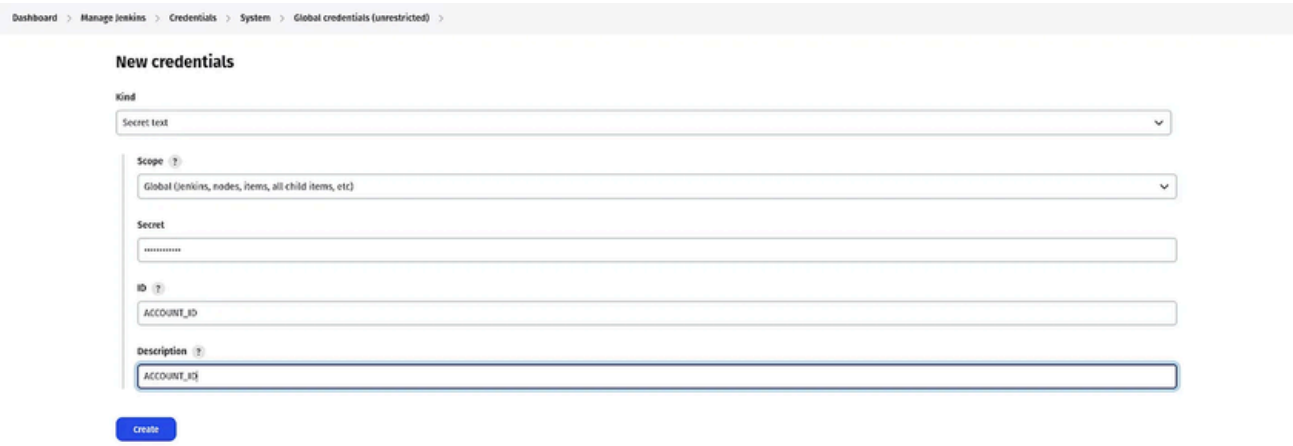
The procedure is the same as when we added the AWS credential secret:

Sonar secret(pass sonar token)



Github PAT(Learn about Gtihub PAT)

Account id(get aws account id)



frontend ECR repo (write in secret “frontend”)

backend ECR repo (write in secret “backend”)

The NVD API key is used to download the security-related database for our dependency check scan. Once you have obtained the API key, save it securely as the secret, where the secret value will be the API key itself.

Kubernetes EKS: Containerize Application

1) Backend Dockerfile

To containerize our Node.js backend, we will create a Dockerfile using the official Node.js 14 image as the base. First, we set the working directory inside the container to **/usr/src/app** to keep everything organized. Then, we copy the **package.json** files and install the required dependencies using **npm install**.

Once the dependencies are installed, we **copy** the rest of the application files into the container. Finally, we use the **CMD** instruction to start the application with the **npm start** command when the container runs.

Dockerfile in my case in EKSNodejs/Backend

```
FROM node:14
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "index.js"]
```

2) Frontend Dockerfile

We will build this Dockerfile using a straightforward approach to set up and run a Node.js application. In this Dockerfile, we start with the Node.js 14 base image and set the working directory. We then copy the **package.json** files and install the necessary dependencies using **npm install**.

After installing dependencies, we **copy** the rest of the application files into the container. Finally, we run the application using the **npm start** command. This approach ensures that all required dependencies are available while keeping the setup simple.

Dockerfile in my case in EKSNodejs/Frontend

```
FROM node:14
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
CMD [ "npm", "start" ]
```

Kubernetes EKS: Setup Secure CI/CD pipeline

Jenkins will be used to deploy containers to ECR registries whenever there are updates to the frontend or backend code, and to update the Kubernetes YAML files. For this, two pipelines—frontend and backend—have been created. To set this up, go to the Jenkins Dashboard → New Item → give your pipeline a name → select Pipeline and click OK.

For the frontend pipeline, navigate to the Frontend pipeline, then copy the entire code into your pipeline script section.

The first block to focus on is the environment block, which pulls data from credentials and stores it as variables for the pipeline. Jenkins needs to know the branch and repository being used, so modify this based on your Git repository structure.

```
peline {
  agent any
  tools {
      jdk 'jdk'
      nodejs 'nodejs'
  }
  environment  {
      NVD_API_KEY = credentials('NVD_API_KEY') // Replace with the ID you used
      SCANNER_HOME=tool 'sonar-scanner'
      AWS_ACCOUNT_ID = credentials('ACCOUNT_ID')
      AWS_ECR_REPO_NAME = credentials('ECR_REP02')
      AWS_DEFAULT_REGION = 'us-east-1'
      REPOSITORY_URI = "${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com/"
  }
  stages {
      stage('Cleaning Workspace') {
          steps {
              cleanWs()
          }
      }
      stage('Checkout from Git') {
          steps {
              git branch: 'main',  url: 'https://github.com/MichaelRobotics/Kubernetes.git'
          }
      }
```

In the SonarQube Analysis stage, the dir block should specify the path to your backend code, as set in the Checkout from Git stage. This adjustment is required in most subsequent steps. Also, provide the SonarQube project name for the frontend, which you configured during the SonarQube setup.

```
  }
  stage('Sonarqube Analysis') {
      steps {
          dir('EKSNodejs/Backend') {
              withSonarQubeEnv('sonar-server') {
                  sh ''' $SCANNER_HOME/bin/sonar-scanner \
                  -Dsonar.projectName=EKSNodejs \
                  -Dsonar.projectKey=EKSNodejs '''
              }
          }
      }
  }
```

For all pipeline stages involving security tasks (e.g., OWASP checks and Trivy scans), ensure that you provide the backend directory path as mentioned earlier.

```
    }
    stage('OWASP Dependency-Check Scan') {
        steps {
            dir('EKSNodejs/Backend') {
                dependencyCheck additionalArguments: '--scan ./ --disableYarnAudit --disableNodeAudit --nvdA
                dependencyCheckPublisher pattern: '**/dependency-check-report.xml'
            }
        }
    }
    stage('Trivy File Scan') {
```

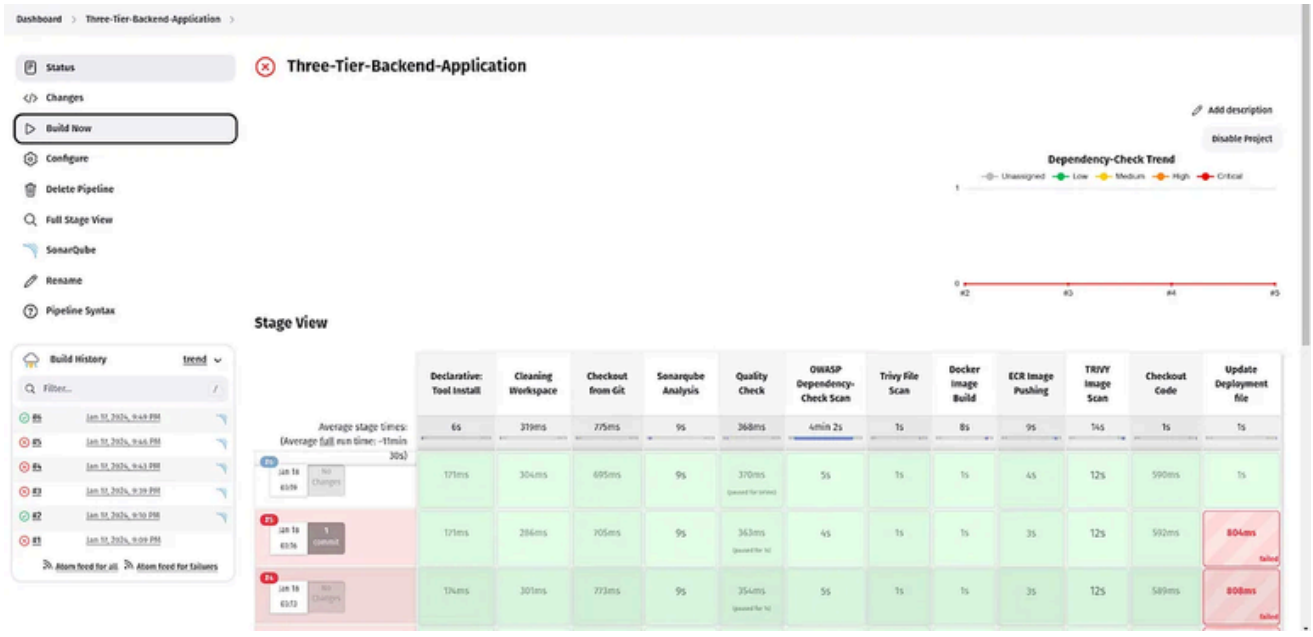
```
}
stage("TRIVY Image Scan") {
    steps {
        sh 'trivy image ${REPOSITORY_URI}${AWS_ECR_REPO_NAME}:${BUILD_NUMBER} > trivyimage.txt'
    }
}
stage('Checkout Code') {
```

The last step is to update the deployment with the new images.

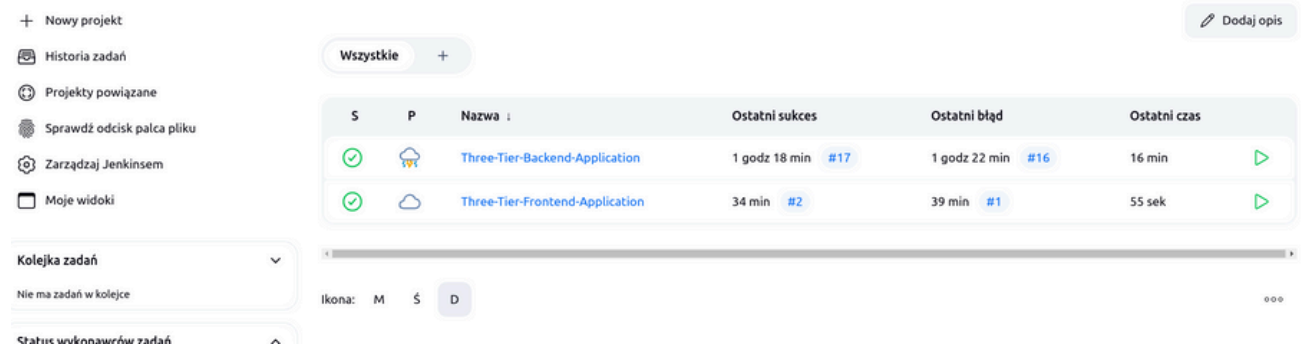
```
}
stage('Checkout Code') {
    steps {
        git branch: 'main', url: 'https://github.com/MichaelRobotics/Kubernetes.git'
    }
}
stage('Update Deployment file') {
    environment {
        GIT_REPO_NAME = "Kubernetes"
        GIT_USER_NAME = "MichaelRobotics"
    }
    steps {
        dir('EKSNodejs/k8s/Backend') {
            withCredentials([string(credentialsId: 'github', variable: 'GITHUB_TOKEN')]) {
                sh '''
                    git config user.email "fotografiaartyzmi01@gmail.com"
                    git config user.name "MichaelRobotics"
                    BUILD_NUMBER=${BUILD_NUMBER}
                    echo $BUILD_NUMBER
                    imageTag=$(grep -oP '(?<=backend:)[^ ]+' deployment.yaml)
                    echo $imageTag
                    sed -i "s/${AWS_ECR_REPO_NAME}:${imageTag}/${AWS_ECR_REPO_NAME}:${BUILD_NUMBER}/" deployment.yaml
                    git add deployment.yaml
                    git commit -m "Update deployment Image to version \${BUILD_NUMBER}"
                    git push https://${GITHUB_TOKEN}@github.com/${GIT_USER_NAME}/${GIT_REPO_NAME} HEAD:main
                '''
            }
        }
    }
}
```

Setup the checkout as you did at the beginning. Change the Git repository name and username. Provide the path to your Kubernetes deployment files. Finally, modify the Git config with git config user.email, git config user.name, and HEAD:<your branch name> (for example, main in my case).

After everything is correctly set, run your pipeline:



Fix problems if you had any and voila, frontend pipeline is setup. As you see, image up there is from backend pipeline. Create new pipeline and follow all steps but with backend pipeline code.



After all needed work, you should have both pipeline configured.

Kubernetes EKS: Setup Prometheus & Grafana observability stack

Add prometheus repo

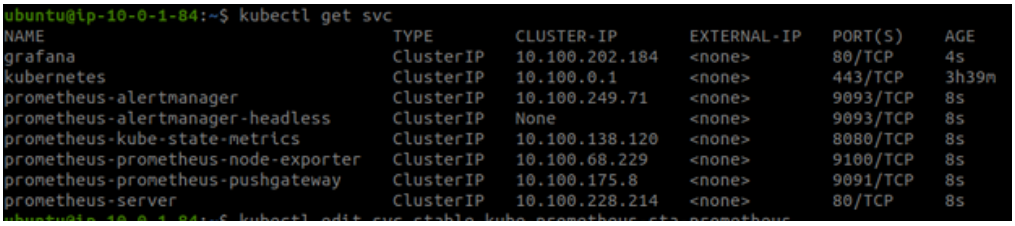
```
helm repo add stable https://charts.helm.sh/stable
```

install prometheus

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm install prometheus prometheus-community/prometheus
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
helm install grafana grafana/grafana
```

get all svc

```
kubectl get svc
```



now edit prometheus-server service, we need it to be LoadBalancer type so we can acces its address from outside of ec2 using url:

```
kubectl edit svc prometheus-server
```

```
    port: 9090
    protocol: TCP
    targetPort: 9090
  - appProtocol: http
    name: reloader-web
    port: 8080
    protocol: TCP
    targetPort: reloader-web
  selector:
    app.kubernetes.io/name: prometheus
    operator.prometheus.io/name: stable-kube-prometheus-sta-prometheus
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer: {}
```

Do the same with grafana

kubectl edit svc stable-grafana

```
    port: 80
    protocol: TCP
    targetPort: 3000
  selector:
    app.kubernetes.io/instance: stable
    app.kubernetes.io/name: grafana
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer: {}
```

Its time to connect grafana and prometheus.

Acces prometheus using promethheus-server: <EXTERNAL-IP>:80

Acces grafana using grafana: <EXTERNAL-IP>:80

```
ILTHiQIronozsUy4sNJHY1dvdx6QQ3FJtwHUAa30ubuntu@ip-10-0-1-84:~$ kubectl get svc
NAME                                  TYPE           CLUSTER-IP       EXTERNAL-IP                                                                PORT(S)          AGE
grafana                               LoadBalancer   10.100.70.180    afbfe3fe6e219401891535cdb6dcf6de-1455594665.us-east-1.elb.amazonaws.com    80:31808/TCP     50m
kubernetes                            ClusterIP      10.100.0.1       <none>                                                                     443/TCP          104m
prometheus-alertmanager               ClusterIP      10.100.152.252   <none>                                                                     9093/TCP         50m
prometheus-alertmanager-headless      ClusterIP      None             <none>                                                                     9093/TCP         50m
prometheus-kube-state-metrics         ClusterIP      10.100.40.187    <none>                                                                     8080/TCP         50m
prometheus-prometheus-node-exporter   ClusterIP      10.100.63.98     <none>                                                                     9100/TCP         50m
prometheus-prometheus-pushgateway     ClusterIP      10.100.57.95     <none>                                                                     9091/TCP         50m
prometheus-server                     LoadBalancer   10.100.13.59     aba8a0771db3b4f66b730426108fbc17-1877728130.us-east-1.elb.amazonaws.com    80:30140/TCP     50m
```

then retrieve grafana credentials:

kubectl get secret grafana -o jsonpath="{.data.admin-user}" | base64 --decode

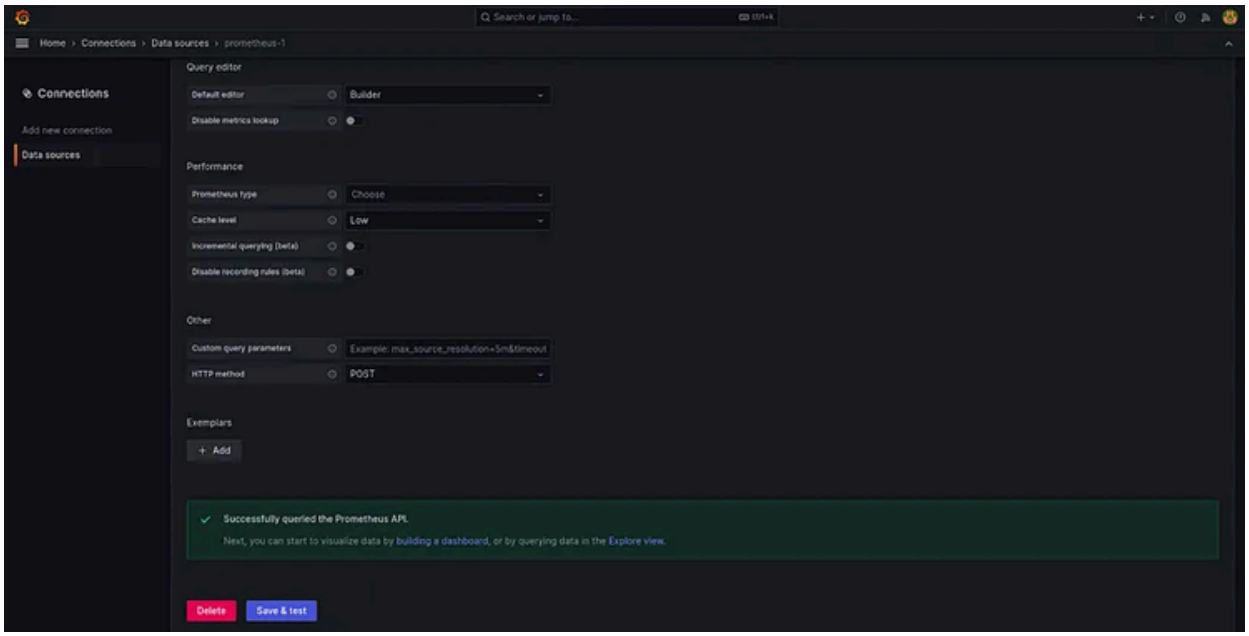
echo ""

kubectl get secret grafana -o jsonpath="{.data.admin-password}" | base64 --decode

```
ubuntu@ip-10-0-1-84:~$ kubectl get secret grafana -o jsonpath="{.data.admin-user}" | base64 --decode
echo ""
kubectl get secret grafana -o jsonpath="{.data.admin-password}" | base64 --decode
admin
ILTHiQIronozsUy4sNJHY1dvdx6QQ3FJtwHUAa30ubuntu@ip-10-0-1-84:~$
```

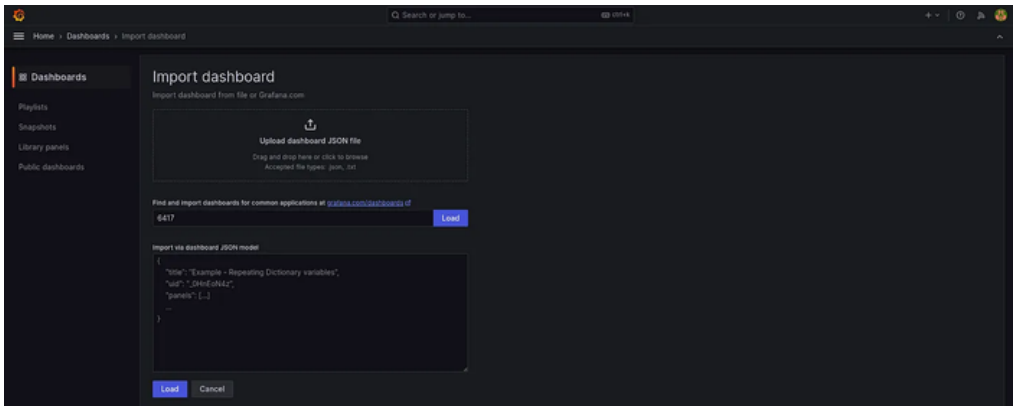
Time to conenct prometheus to grafana. Log into grafana, go data sources->select prometheus-> in connection paste prometheus url <EXTERNAL-IP>:80-> Click save & test

If the URL is correct, then you will see a green notification



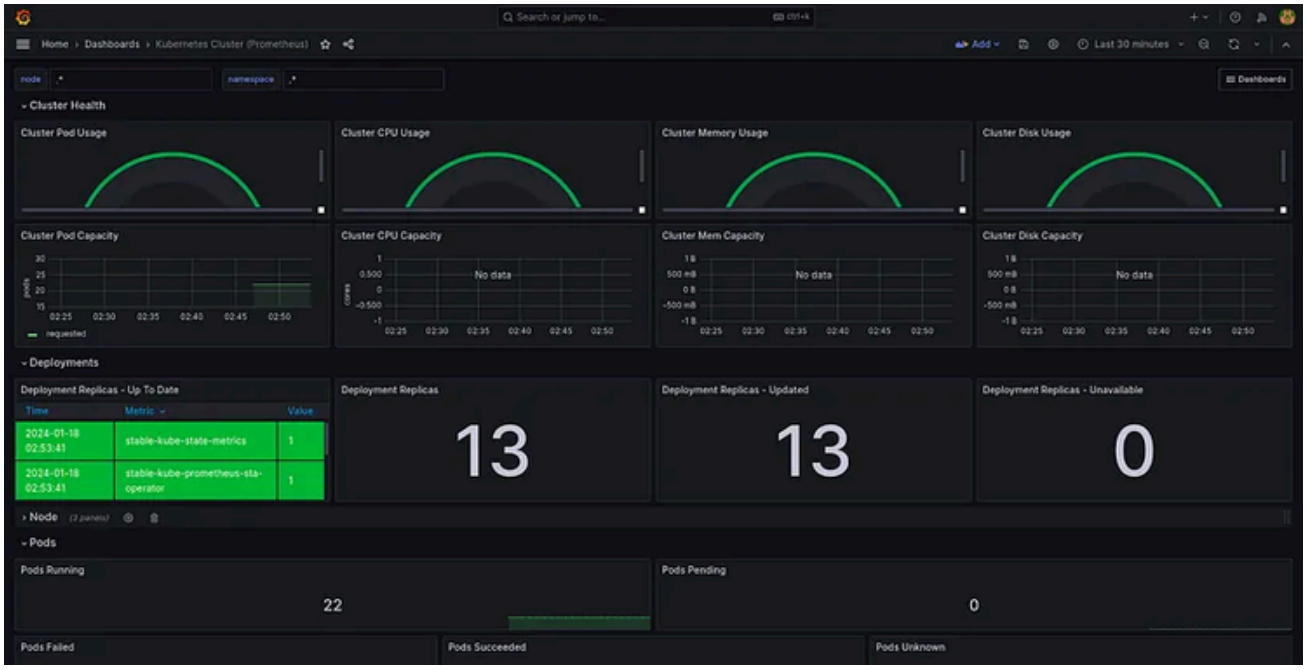
Now we will create Dashboard. Go Dashboards->New->Import->Provide “6417”-> Click Load

Select the data source



Select the data source to prometheus-1 (this what we created earlier)

Here you go. Now you can access your Dashboard:



Kubernetes EKS: Configure ArgoCD to deploy App, setup custom domain

It's time to set up GitOps with ArgoCD. Navigate to the ArgoCD Dashboard open the ArgoCD URL in your browser. Connect the Repository:

Click "Connect Repo" and choose the "HTTPS" option.
Provide the following details for your main repository:

Type: git
Project: default
Repository URL: https://github.com/MichaelRobotics/Kubernetes.git (replace with your repository path)
Username: <Your GitHub username>
Password: <Your GitHub Personal Access Token (PAT)>

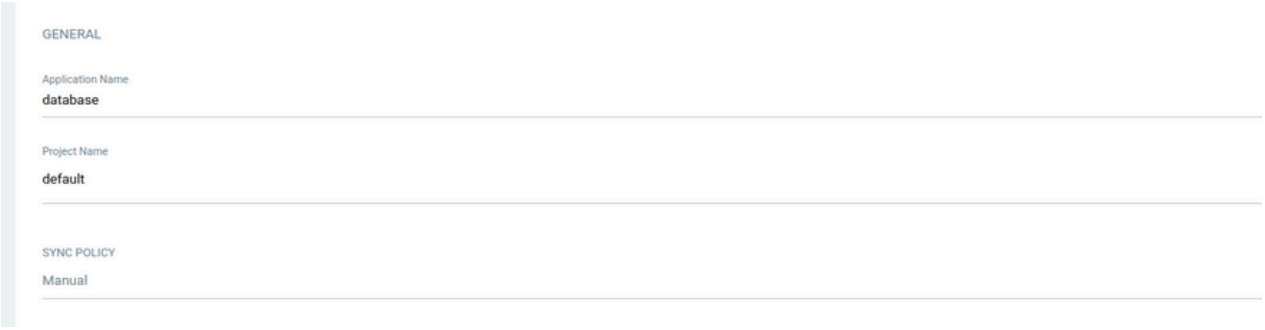
Once the connection is successful, we will create an ArgoCD project. This project will link to your repository and monitor YAML files within it.

ArgoCD will compare the actual state of your Kubernetes cluster with the state defined in your GitHub repository. If any discrepancies are found, ArgoCD will automatically attempt to sync the cluster to match the desired state as described in your repository.

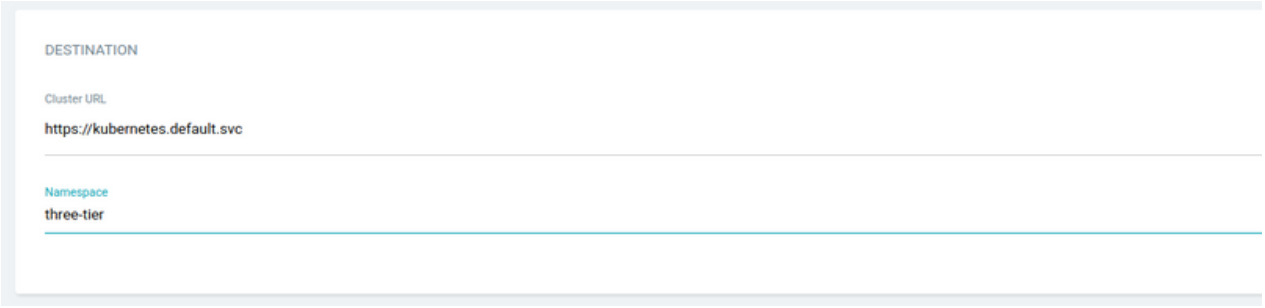
Repository Structure

Your GitHub repository contains four folders representing different tiers and resources:
Frontend, Backend, Database, Ingress

First, we'll connect the directory containing the Database YAML files to ArgoCD. Navigate to Create Application and fill in the following details: application name and project name. Set the sync policy to either manual or automatic. If you choose manual, ArgoCD will wait for your approval before syncing the Kubernetes cluster with the GitHub repository.

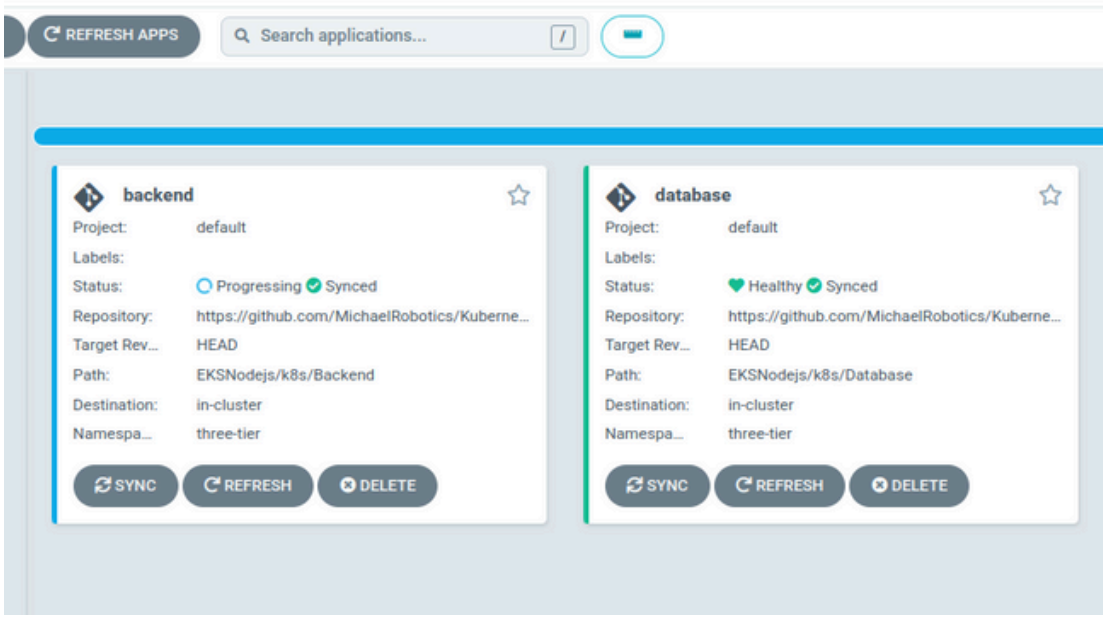


Set the Repository URL to your GitHub repository, set the Revision to HEAD, and specify the path to the folder containing the Database YAML files in the Path field. This will ensure ArgoCD syncs the correct directory from your repository.

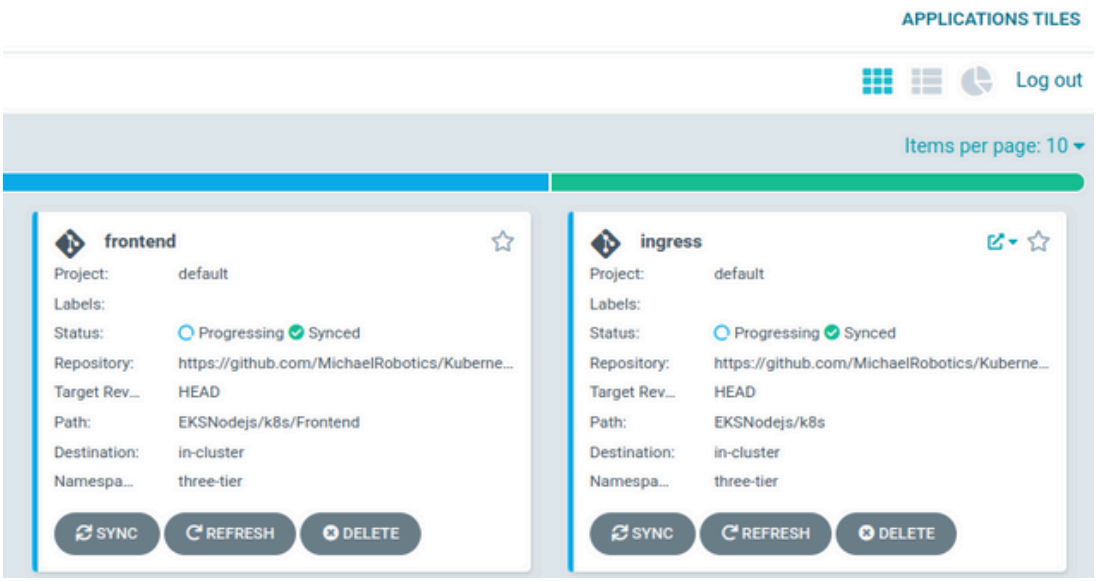


Next, set the Cluster URL to your Kubernetes cluster and specify the Namespace where changes will be monitored. Once everything is configured, click CREATE to finalize the application setup.

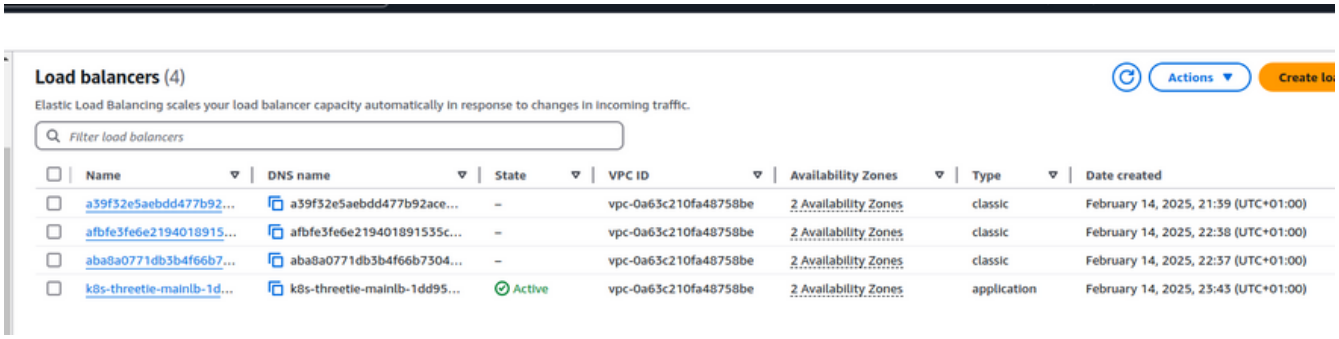
Follow the same process for the Backend, Frontend, and Ingress directories. For each, set the repository URL, revision to HEAD, and the appropriate folder path. Once done, you should see all projects listed in ArgoCD, along with their current sync status and state.



As you can see, the Backend, Frontend, and Ingress applications are still syncing, while the Database has synced successfully. Now, you can go to your Kubernetes cluster and verify if its state matches the configuration defined in your GitHub repository.



Finally, we will set up our domain. The Ingress resource deployed via ArgoCD has created a LoadBalancer. You can retrieve its DNS name from AWS to configure your domain accordingly. This will allow external traffic to reach your services.



The LoadBalancer DNS name is k8s-threrier[...]. In the Ingress resource, we've specified the host with a domain like .epicdns.me. Here's how to complete the domain setup:

Use a domain provider like Cloudflare.

Add a subdomain (e.g., myepicdomain.epicdns.me).

Associate the LoadBalancer DNS with this subdomain.

This setup will allow your subdomain (e.g., myepicdomain.epicdns.me) to route traffic to your Kubernetes services via the LoadBalancer DNS.

EKSNodejs > k8s > ! ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: mainlb
  namespace: three-tier
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80}]'
    # alb.ingress.kubernetes.io/subnets: "subnet-0aa439b4ddafcca10,subnet-0b95df318219145ca,subnet-0ccca36474902829a"
spec:
  ingressClassName: alb
  rules:
    - host: .epicdns.me
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: api
```

After some time, you should be able to acces your app from internet. Enjoy!

common troubleshooting

1) Secure Jenkins CI/CD Pipeline Fails

Cause: Incorrect Jenkins pipeline configuration or missing Kubernetes/EKS credentials.
Solution: Check pipeline logs for errors and verify Kubernetes credentials using kubectl config use-context <eks-cluster>. Ensure proper IAM roles are configured.

2) GitOps Sync Issues in ArgoCD

Cause: GitHub repo and Kubernetes cluster configuration mismatch.
Solution: Ensure ArgoCD is synced with the GitHub repo. Check sync status in ArgoCD UI or CLI and use argocd app sync <app-name> to initiate sync.

3) Terraform Deployment Errors

Cause: Misconfigured Terraform variables or missing IAM roles.
Solution: Validate Terraform files with terraform validate and run terraform plan to identify missing variables or misconfigurations.

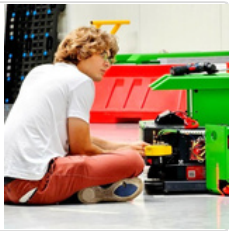
4) Grafana Not Displaying Metrics

Cause: Misconfigured Prometheus metrics in Grafana.
Solution: Verify Prometheus is scraping correct targets and check the Prometheus data source settings in Grafana to ensure it's correctly pointing to the Prometheus instance.

5) Check my Kubernetes Troubleshooting series:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27

Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

https://kubernetes.io/docs/setup/

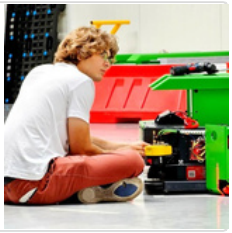
Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!