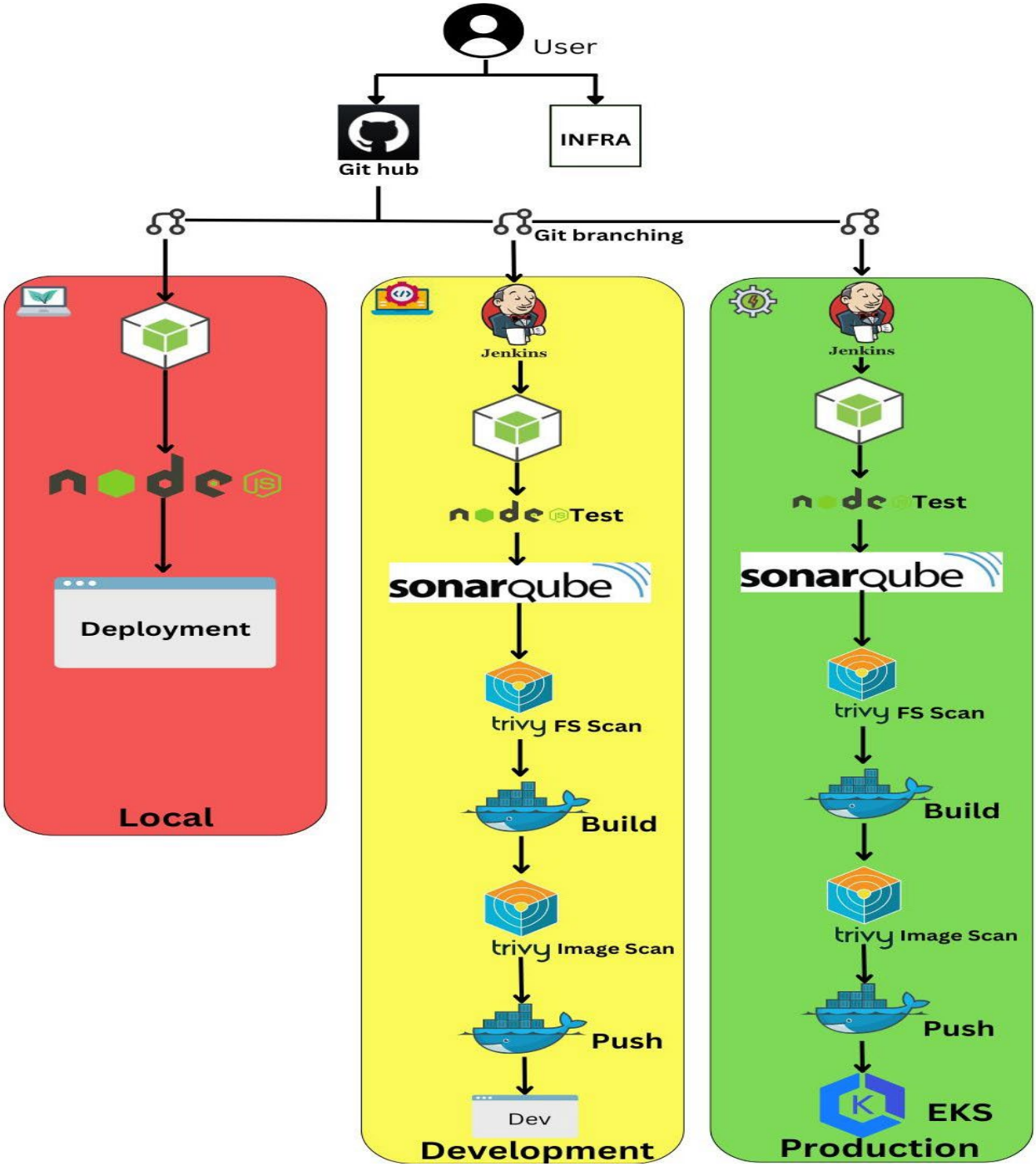
DevOps

Three Tier Full Stack Project

Introduction

The "Three Tier Full Stack Project" is a comprehensive endeavor designed to implement a multi-tiered architecture using modern DevOps practices. This project involves setting up a continuous integration and continuous deployment (CI/CD) pipeline with tools such as **Jenkins, Docker**, **SonarQube**, **Trivy** and **AWS EKS**.

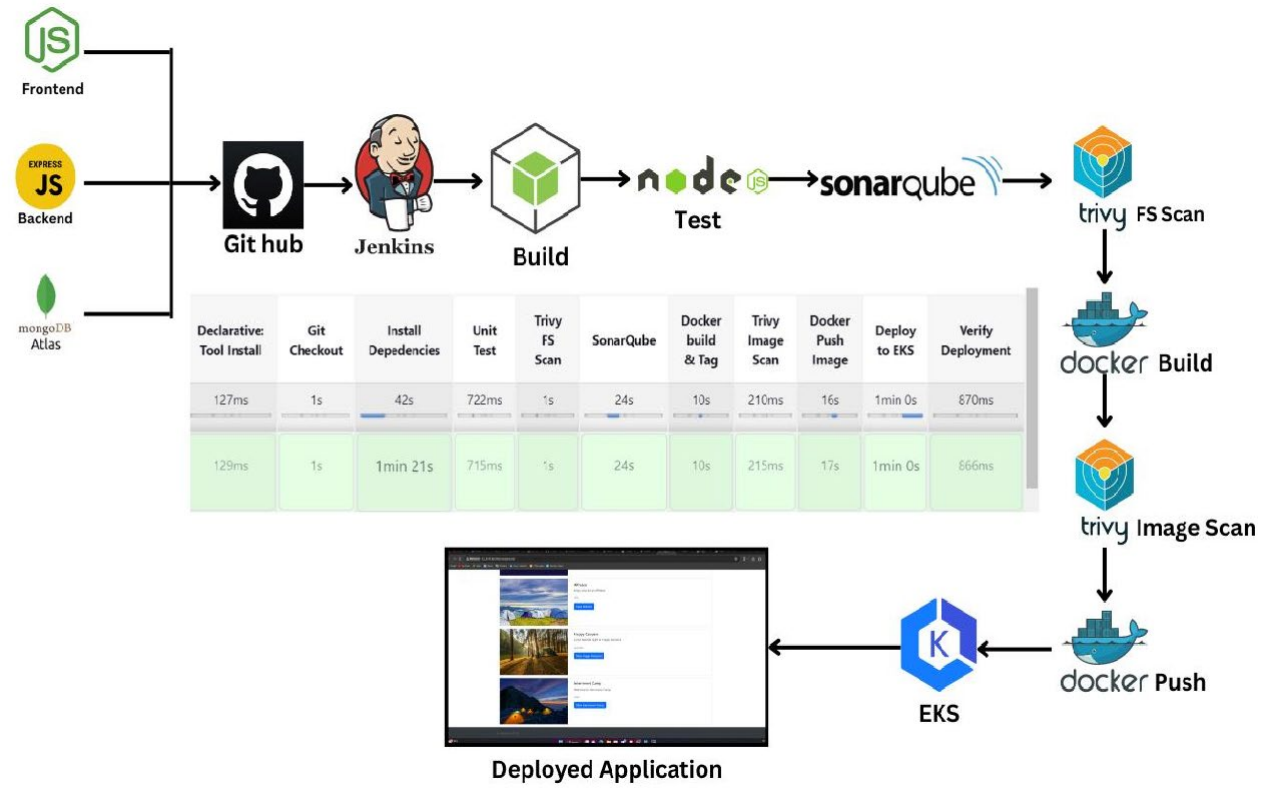


Project Overview

The project is divided into several key components:

1. **Setting up Jenkins**: Continuous Integration server setup.
2. **Installing Docker**: Containerization tool installation for environment consistency.
3. **Setting up SonarQube**: Code quality and security analysis.
4. **Setting up EKS**: Managed Kubernetes service on AWS for container orchestration.
5. **Pipeline and Deployments**: Automating builds, tests, and deployments.

Architechture



Git hub Repository Link-

https://github.com/Bijan1235/3-Tier-Full-Stack.git

Environments

1. Local Environment

Purpose: For initial development and testing on a developer's machine.

Workflow:

- Developers clone the repository from GitHub.
- Application runs using Node.js locally.
- Code is tested and debugged before pushing to the shared repository.

2. Development Environment

Purpose: For integrating changes, running automated tests, and security scans.

Workflow:

- Code is fetched from GitHub via Jenkins.
- Tests are executed using Node.js.
- Code quality is checked with SonarQube.
- Security scans are performed with Trivy.
- Docker image is built and scanned.
- Application is deployed to the development server.

3. Production Environment

Purpose: For deploying the stable and tested application to the live environment.

Workflow:

- Code goes through the same testing and scanning steps as the development environment.
- The Docker image is pushed to the production registry.
- Application is deployed to AWS EKS for production use.

Environment Variables Setup

Environment variables are crucial for configuring the application across different environments such as local development, development, and production. They allow sensitive information and configuration settings to be managed securely and separately from the application's codebase.

To set up the required environment variables for your project, you need to obtain the values from the respective services you are using. Below is a detailed guide on how to obtain each of the necessary environment variables:

1. Cloudinary Configuration

Cloudinary is a cloud-based image and video management service. To get the required Cloudinary credentials:

1. **Sign Up or Log In to Cloudinary**:
    - Go to Cloudinary and sign up for a free account or log in if you already have one.
2. **Get Cloudinary API Credentials**:
    - Once logged in, navigate to the Cloudinary Dashboard.
    - Your Cloudinary API credentials, including the **cloud_name**, **api_key**, and **api_secret**, are available in the dashboard under the **Account Details** section.

Example Values:

```
CLOUDINARY_CLOUD_NAME=dj0lkx4sa
CLOUDINARY_KEY=686681518851598
CLOUDINARY_SECRET=hSGeIYLq2FtaokM80E4tlGnoVYY
```

2. Mapbox Token

Mapbox is a service for custom online maps. To get your Mapbox token:

1. **Sign Up or Log In to Mapbox**:
    - Go to Mapbox and sign up for an account or log in if you already have one.

2. **Get Mapbox Access Token**:
    - Once logged in, navigate to your Account page.
    - Under the **Access Tokens** section, you can create a new token or use an existing one.

Example Value:

MAPBOX_TOKEN=sk.eyJ1Ijoic2h1YmhhbTIxMjEiLCJhIjoiY2x3Z2ZreXIyMDBnYTJpb3dlcTFqY2ttNiJ9.T6SRpJMnV2ZpahT_l0N61A

3. MongoDB Database URL

MongoDB Atlas is a cloud-based database service. To get your MongoDB connection string:

1. **Sign Up or Log In to MongoDB Atlas**:
    - Go to MongoDB Atlas and sign up for a free account or log in if you already have one.
2. **Create a New Cluster**:
    - If you don't have a cluster, create a new one by following the on-screen instructions.
3. **Get MongoDB Connection String**:
    - Once your cluster is set up, go to the **Clusters** view.
    - Click the **Connect** button for your cluster.
    - Follow the instructions to get the connection string. Make sure to replace **<password>** with your database user password and **<dbname>** with the name of your database.

Example Value:

DB_URL="mongodb+srv://pttnkbjn:xwYU67SnBY1P2JXL@bijan-ds.tkrlvf1.mongodb.net/?retryWrites=true&w=majority&appName=Bijan-DS"

4. Secret Key

The **SECRET** key is used for session management, encryption, or other security purposes in your application. This key should be a strong, unique string.

1. **Generate a Secret Key**:You can generate a secret key using any method that provides a random, secure string. **Example Value**:

SECRET=bijan

Environment Variables:

Once you have obtained all the required values, create a **.env** file in the root directory of your project and add the following lines:

CLOUDINARY_CLOUD_NAME=dcxkeojag

CLOUDINARY_KEY=954893496495264

CLOUDINARY_SECRET=LfNYj1hnJs9uv095D5iKfSTFNOw

MAPBOX_TOKEN=sk.eyJ1IjoiYmlqYW41IiwiYSI6ImNseGhqYjAxMzE2bTMybHF0eGNjN2V6dzQifQ.itÑPybR6dUcdzDQfBZkBTQ

DB_URL="mongodb+srv://pttnkbjn:xwYU67SnBY1P2JXL@bijan-ds.tkrlvf1.mongodb.net/?retryWrites=true&w=majority&appName=Bijan-DS"

SECRET=bijan

Phase-1: Local Deployment

To deploy your application locally following the steps you provided, you'll need to execute the following commands on your T2.Medium Ubuntu machine:

1. Connect to the machine

2. Clone the repository:

```
git clone <repository_url>
cd <repository_directory>
```

3. Install Node.js using NVM:

Install NVM (Node Version Manager)

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

Execute below commands

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"
```

download and install Node.js

```
nvm install 21
```

verifies the right Node.js version is in the environment

```
node -v
```

verifies the right NPM version is in the environment

```
npm -v
```

4. Obtain API keys:

- Create an account on Cloudinary and obtain your cloud name, API key, and secret.
- Create an account on Mapbox and obtain your public access token.
- Sign up for MongoDB Atlas and create a database. Retrieve your connection URL.

5. Create a .env file:

`vi .env` file in the project directory

Add the following lines to the file and replace placeholders with your actual values:

CLOUDINARY_CLOUD_NAME=[Your Cloudinary Cloud Name]

CLOUDINARY_KEY=[Your Cloudinary Key]

CLOUDINARY_SECRET=[Your Cloudinary Secret]

MAPBOX_TOKEN=[Your Mapbox Token]

DB_URL=[Your MongoDB Atlas Connection URL]

SECRET=[Your Chosen Secret Key]

6. Install project dependencies:

```
npm install
```

7. Start the application:

```
npm start
```

8. Access the app: Open a web browser and navigate

to http://VM_IP:3000 (replace VM_IP with the IP address of your Ubuntu machine).

Phase-2: Development Environment Deployment

Launch Virtual Machine using AWS EC2

Here is a detailed list of the basic requirements and setup for the EC2 instance i have used for running Jenkins, including the specifics of the instance type, AMI, and security groups.

EC2 Instance Requirements and Setup:

1. Instance Type

- Instance Type: `t2.large`

  - vCPUs: 2

  - Memory: 8 GB

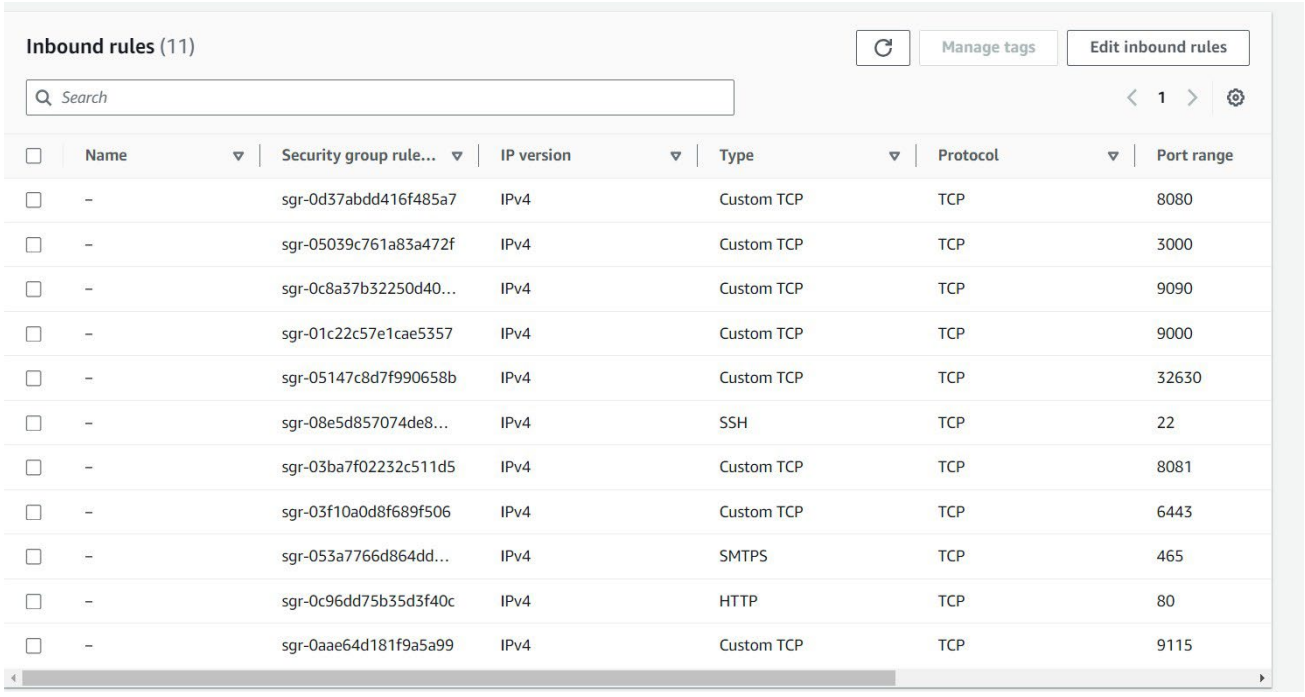
  - Network Performance: Moderate

2. Amazon Machine Image (AMI)

- AMI: Ubuntu Server 20.04 LTS (Focal Fossa)

3. Security Groups

Security groups act as a virtual firewall for your instance to control inbound and outbound traffic.



Name	Security group rule...	IP version	Type	Protocol	Port range
-	sgr-0d37abdd416f485a7	IPv4	Custom TCP	TCP	8080
-	sgr-05039c761a83a472f	IPv4	Custom TCP	TCP	3000
-	sgr-0c8a37b32250d40...	IPv4	Custom TCP	TCP	9090
-	sgr-01c22c57e1cae5357	IPv4	Custom TCP	TCP	9000
-	sgr-05147c8d7f990658b	IPv4	Custom TCP	TCP	32630
-	sgr-08e5d857074de8...	IPv4	SSH	TCP	22
-	sgr-03ba7f02232c511d5	IPv4	Custom TCP	TCP	8081
-	sgr-03f10a0d8f689f506	IPv4	Custom TCP	TCP	6443
-	sgr-053a7766d864dd...	IPv4	SMTPS	TCP	465
-	sgr-0c96dd75b35d3f40c	IPv4	HTTP	TCP	80
-	sgr-0aae64d181f9a5a99	IPv4	Custom TCP	TCP	9115

After Launching your Virtual machine ,**SSH** into the Server.

Installing Jenkins on Ubuntu

Execute these commands on Jenkins Server

```
#!/bin/bash
# Install OpenJDK 17 JRE Headless
sudo apt install openjdk-17-jre-headless -y
# Download Jenkins GPG key
sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
# Add Jenkins repository to package manager sources
echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
/etc/apt/sources.list.d/jenkins.list > /dev/null
# Update package manager repositories
sudo apt-get update
# Install Jenkins
sudo apt-get install jenkins -y
```

Save this script in a file, for example, install_jenkins.sh, and make it executable using:

```
chmod +x install_jenkins.sh
```

Then, you can run the script using:

```
./install_jenkins.sh
```

This script will automate the installation process of OpenJDK 17 JRE Headless and Jenkins.

Install Docker Using this command

Sudo apt-get install docker.io

Run this command to give access to docker

sudo chmod 666 /var/run/docker.sock

Install Trivy on Jenkins Server

Install Trivy for file System Scan

sudo apt-get install wget apt-transport-https gnupg lsb-release

wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | sudo apt-key add -

echo deb https://aquasecurity.github.io/trivy-repo/deb $(lsb_release -sc) main | sudo tee -a /etc/apt/sources.list.d/trivy.list

sudo apt-get update

sudo apt-get install trivy

Create another Virtual Machine for SonarQube Scanner

You can use “t2.meduim” for sonarqube server

Steps to Run SonarQube using Docker:

- SSH into the server .
- Install docker using the above command.
- Run the following command to run SonarQube.
  docker run -d –name sonarqube -p 9000:9000 sonarqube:lts-community
- Run this command to check your Sonarqube server is running or not.
  Docker ps
- Connect to http://VM_IP:9000 (replace VM_IP with the IP address of your Ubuntu machine) your Sonarqube server is ready.

Configure Jenkins

Access Jenkins Dashboard:

Open a web browser and navigate to your Jenkins instance (e.g., http://your-instance-public-dns:8080).

Log in with your Jenkins credentials. (cat address provided on Jenkins)

Install Plugins:

-Go to Manage Jenkins > Manage Plugins.

-Click on the Available tab.

Search for and install the following plugins:

- **Docker**: Enables Jenkins to use Docker containers.
- **Nodejs**: Required dependency for Nodejs Applications.
- **Sonar Scanner**: For Scanning Vulnerabilities.
- **Docker Pipeline**: Allows Jenkins to use Docker containers in pipeline jobs.
- **Kubernetes**: Provides support for Kubernetes in Jenkins.
- **Kubernetes CLI**: Allows Jenkins to interact with Kubernetes clusters.

Configuring NodeJS Plugin

1. **Manage Jenkins**: Go to "Manage Jenkins".
2. **Global Tool Configuration**: Click on "Global Tool Configuration".
3. **NodeJS**:
    - Scroll down to the "NodeJS" section.
    - Click "Add NodeJS".
    - Provide a name (e.g., **node14**).
    - Select the version of NodeJS you want to install.
    - Optionally, check "Install automatically" to let Jenkins handle the installation.
    - Save the configuration.

Configuring SonarQube Scanner Plugin

1. **Manage Jenkins**: Go to "Manage Jenkins".
2. **Global Tool Configuration**: Click on "Global Tool Configuration".
3. **SonarQube Scanner**:
    - Scroll down to the "SonarQube Scanner" section.
    - Click "Add SonarQube Scanner".
    - Provide a name (e.g., **Sonar scanner**).
    - Optionally, check "Install automatically" to let Jenkins handle the installation.
    - Save the configuration.
4. **Manage Jenkins**: Go back to "Manage Jenkins".
5. **Configure System**:
    - Scroll down to the "SonarQube servers" section.
    - Click "Add SonarQube".
    - Provide a name for the server (e.g., **SonarQube**).
    - Set the "Server URL" to the URL of your SonarQube instance.
    - Add a "Server Authentication Token".

Creating a Token on SonarQube

1. **Log in to SonarQube**: Open your SonarQube instance in a web browser and log in.
2. **My Account**: Click on your user profile at the top-right corner and select "My Account".
3. **Security**: Navigate to the "Security" tab.
4. **Generate Token**: Under "Generate Tokens", provide a name for the token (e.g., **JenkinsToken**).
5. **Generate**: Click on "Generate" and copy the token.

Adding SonarQube Token to Jenkins

1. **Manage Jenkins**: Go to "Manage Jenkins".
2. **Configure System**: Scroll to the "SonarQube servers" section.
3. **Add Token**:
    - Under the "Server Authentication Token" section, click "Add" next to "Credentials".
    - Select "Jenkins" and then "Secret text".
    - Paste the token you copied from SonarQube.
    - Provide an ID (e.g., **sonarqube-token**).
    - Save the credentials.
    - Select the newly added token from the dropdown list.

Configuring Docker Plugin

1. **Manage Jenkins**: Go to "Manage Jenkins".
2. **Global Tool Configuration**: Click on "Global Tool Configuration".
3. **Docker**:
    - Scroll down to the "Docker" section.
    - Click "Add Docker Tool".
    - Provide a name (e.g., **docker**).
    - Optionally, check "Install automatically" to let Jenkins handle the installation.
    - Save the configuration.

Create a Pipeline Job

Create a Pipeline job and name it as – "**Dev-env-3tier**"

Give the maximum builds as 2.

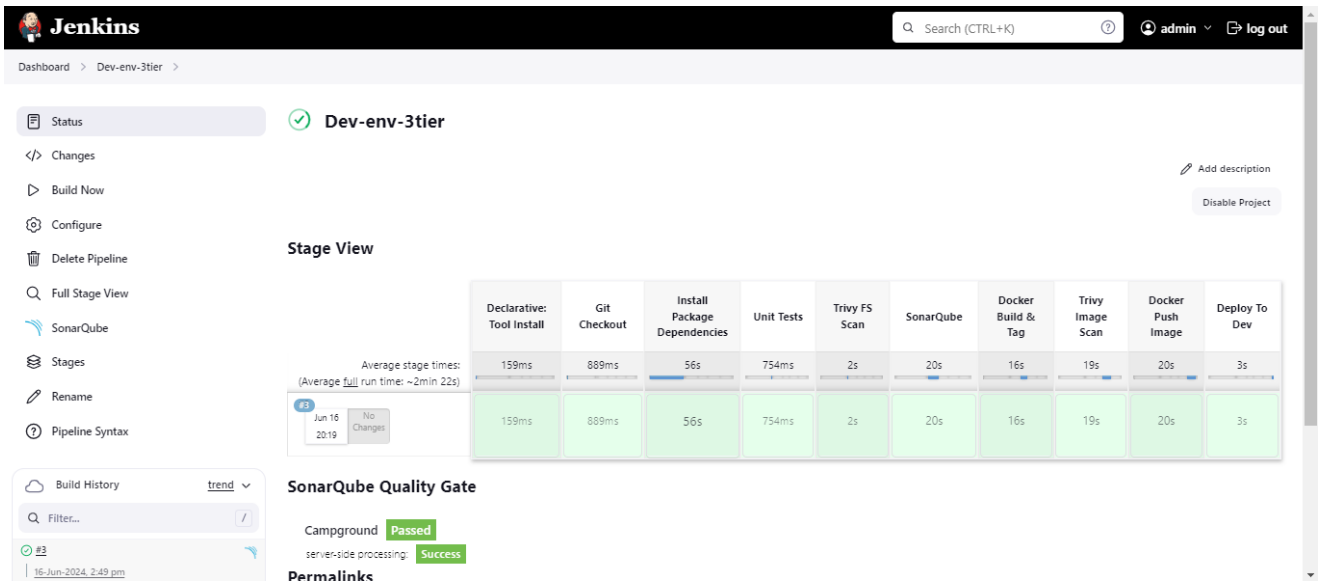
Pipeline :

```
pipeline {
   agent any

   tools {
      nodejs 'node21'
   }

   environment {
      SCANNER_HOME= tool "sonar-scanner"
   }

   stages {
      stage('Git Checkout') {
         steps {
            git branch: 'main', credentialsId: 'git-cred', url: 'https://github.com/Bijan1235/3-Tier-Full-Stack.git'
         }
      }
      stage('Install Depedencies') {
         steps {
            sh "npm install"
         }
      }
      stage('Unit Test') {
         steps {
            sh "npm test"
         }
```

```
        }
        stage('Trivy FS Scan') {
            steps {
                sh "trivy fs --format table -o fs-report.html ."
            }
        }
        stage('SonarQube') {
            steps {
                withSonarQubeEnv('sonar') {
                    sh " $SCANNER_HOME/bin/sonar-scanner -Dsonar.projectKey=Campground -
DsonarprojectName=Campground"
                }
            }


        }
        stage('Docker build & Tag') {
            steps {
                script{
                    withDockerRegistry(credentialsId: 'docker-cred1', toolName: 'docker ') {
                        sh "docker build -t bijan9438/camp:latest ."
                    }
                }
            }
        }
        stage('Trivy Image Scan') {
            steps {
                sh "trivy image --format table -o fs-report.html bijan9438/camp:latest"
            }
        }
        stage('Docker Push Image') {
            steps {
```

```
            script{
                withDockerRegistry(credentialsId: 'docker-cred1', toolName: 'docker ') {
                    sh "docker push bijan9438/camp:latest"
                }
            }
        }
    }
    stage('Docker Deploy to Dev') {
        steps {
            script{
                withDockerRegistry(credentialsId: 'docker-cred1', toolName: 'docker ') {
                    sh "docker run -d -p 3000:3000 bijan9438/camp:latest"
                }
            }
        }
    }
  }
}
```

Result



Phase-3: Production Environment Deployment

Install AWS CLI , EKSCTL & KUBECTL on Jenkins Server

AWSCLI

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"

sudo apt install unzip

unzip awscliv2.zip

sudo ./aws/install
```

KUBECTL

```
curl -o kubectl https://amazon-eks.s3.us-west-2.amazonaws.com/1.19.6/2021-01-05/bin/linux/amd64/kubectl

chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin

kubectl version --short --client
```

EKSCTL

```
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/local/bin

eksctl version
```

Save all the script in a file, for example, ctl.sh, and make it executable

using:

```
chmod +x ctl.sh
```

Then, you can run the script using:-→ `./ctl.sh`

Create a user in AWS IAM with any name

Attach Policies to the newly created user.

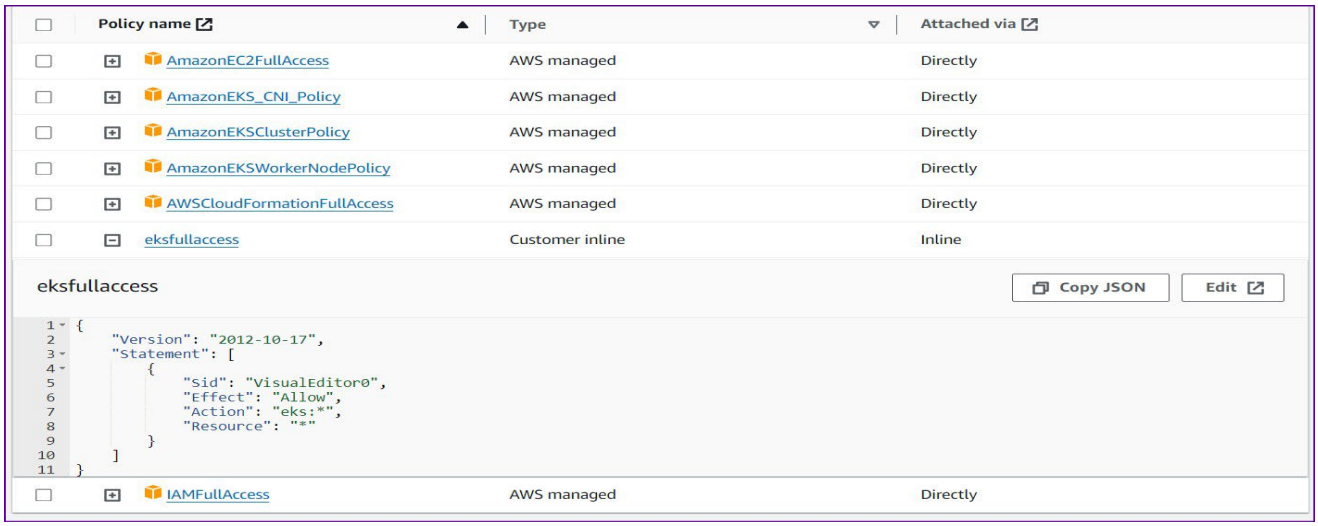
below policies:

- *AmazonEC2FullAccess*
- *AmazonEKS_CNI_Policy*
- *AmazonEKSClusterPolicy*
- *AmazonEKSWorkerNodePolicy*
- *AWSCloudFormationFullAccess*
- *IAMFullAccess*

One more **inline policy** we need to create with content as below:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": "eks:*",
      "Resource": "*"
    }
  ]
}
```

Attach this policy to your user as well.



Once IAM User is created, Create its Secret Access Key and download the **credentials.csv** file .

Run the following command on server to connect to AWS

```
aws configure
```

Provide the Access key and Secret Access key and region. (present in **credentials.csv** file)

Now you are connected to your AWS .

Create EKS Cluster

```
eksctl create cluster --name=EKS-2 \
                --region=ap-south-1 \
                --zones=ap-south-1a,ap-south-1b \
                --without-nodegroup
```

Open ID Connect

```
eksctl utils associate-iam-oidc-provider \
  --region ap-south-1 \
  --cluster EKS-2 \
  --approve
```

Create node Group

```
eksctl create nodegroup --cluster=EKS-2 \
                --region=ap-south-1 \
                --name=node2 \
                --node-type=t3.medium \
                --nodes=3 \
                --nodes-min=2 \
                --nodes-max=3 \
```

```
                --node-volume-size=20 \
                --ssh-access \
                --ssh-public-key=Bijan-Mumbai \
                --managed \
                --asg-access \
                --external-dns-access \
                --full-ecr-access \
                --appmesh-access \
            --alb-ingress-access
```

Make sure to change the name of **ssh-public-Key** with your SSH key.

Run these commands on Server

Create Service Account, Role & Assign that role, And create a secret for Service Account and generate a Token.

Create a file : Vim **svc.yml**

Creating Service Account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins
  namespace: webapps
```

To run the svc.yml : kubectl apply -f svc.yaml

Similarly create a **role.yml** file

Create Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: app-role
  namespace: webapps
rules:
  - apiGroups:
      - ""
      - apps
      - autoscaling
      - batch
      - extensions
```

```
    - policy
    - rbac.authorization.k8s.io
  resources:
    - pods
    - componentstatuses
    - configmaps
    - daemonsets
    - deployments
    - events
    - endpoints
    - horizontalpodautoscalers
    - ingress
    - jobs
    - limitranges
    - namespaces
    - nodes
    - pods
    - persistentvolumes
    - persistentvolumeclaims
    - resourcequotas
    - replicasets
    - replicationcontrollers
    - serviceaccounts
    - services
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

To run the role.yaml file: `kubectl apply -f role.yaml`

Similarly create a **bind.yml** file

Bind the role to service account

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-rolebinding
  namespace: webapps
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: app-role
subjects:
- namespace: webapps
  kind: ServiceAccount
  name: jenkins
```

To run the bind.yaml file: `kubectl apply -f bind.yaml`

Create Token

Similarly create a secret.yml file

```
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: mysecretname
  annotations:
    kubernetes.io/service-account.name: Jenkins
```

To run the secret.yml file: `kubectl apply -f secret.yml -n webapps`

Save the token .

Go to the **pipeline syntax** and select **With Kubernetes:Configure Kubernetes**

1. **Credentials** – Provide the Token that you have saved .
2. **Kubernates Endpoint API**- You can find it in your AWS EKS cluster.
3. **Cluster name**- Provide any name.
4. **NameSpace** – webapps

Click on Generate Syntax.

You will get pipeline syntax :-

```
withKubeCredentials(kubectlCredentials: [[caCertificate: '', clusterName: 'EKS-1', contextName: '', credentialsId: 'k8-token', namespace: 'webapps', serverUrl: 'https://B7C7C20487B2624AAB0AD54DF1469566.yl4.ap-south-1.eks.amazonaws.com']]) {
//block of code
            }
```

Encoding Environment Variables to Base64 for Kubernetes Secrets

When deploying applications to Kubernetes, sensitive information such as environment variables must be stored securely. Kubernetes Secrets is a native way to handle such sensitive data. In the provided YAML manifest files, environment variables are stored as base64 encoded values. Below is a guide on how to encode your environment variables to base64 and include them in your Kubernetes manifest files.

Steps to Encode Environment Variables to Base64

1. **Prepare Your Environment Variables:**
    - CLOUDINARY_CLOUD_NAME: dcxkeojag
    - CLOUDINARY_KEY: 954893496495264
    - CLOUDINARY_SECRET: LfNYj1hnJs9uv095D5iKfSTFNOw
    - MAPBOX_TOKEN: sk.eyJ1IjoiYmlqYW41IiwiYSI6ImNseGhqYjAxMzE2bTMybHF0eGNjN2V6dzQifQ.itNPybR6dUcdzDQfBZkBTQ
    - DB_URL: mongodb+srv://pttnkbjn:xwYU67SnBY1P2JXL@bijan-ds.tkrlvf1.mongodb.net/?retryWrites=true&w=majority&appName=Bijan-DS
    - SECRET: bijan
2. **Encode Each Value to Base64:** You can encode these values using the base64 command-line tool or an online base64 encoder.

echo -n 'dcxkeojag' | base64 # CLOUDINARY_CLOUD_NAME echo -n

echo -n '954893496495264' | base64 # CLOUDINARY_KEY echo -n

echo -n 'LfNYj1hnJs9uv095D5iKfSTFNOw' | base64 # CLOUDINARY_SECRET echo -n

echo -n 'sk.eyJ1IjoiYmlqYW41IiwiYSI6ImNseGhqYjAxMzE2bTMybHF0eGNjN2V6dzQifQ.itNPybR6dUcdzDQfBZkBTQ' | base64 # MAPBOX_TOKEN echo -n

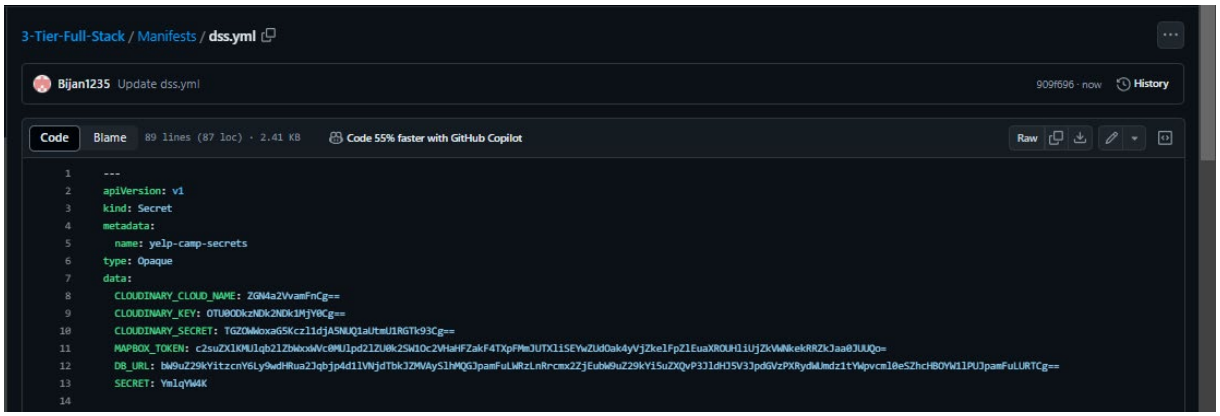
echo -n 'mongodb+srv://pttnkbjn:xwYU67SnBY1P2JXL@bijan-ds.tkrlvf1.mongodb.net/?retryWrites=true&w=majority&appName=Bijan-DS ' | base64 # DB_URL echo -n '

echo -n 'bijan' | base64 # SECRET

The encoded values will look like this:

- CLOUDINARY_CLOUD_NAME: ZGo0bGt4NHNh
- CLOUDINARY_KEY: Njg2NjgxNTE4ODUxNTk4
- CLOUDINARY_SECRET: aFNHZWlZTHEyRnRhb2tNNDBFNG5vVllZ
- MAPBOX_TOKEN: c2suZXlKMUlqb2lZV1JwYW1GcGMzZGhiQ0lzSW1FaU9pSmpiSFl3TnpKemVHNHhaRzR5TW1wd1ltWm1OSFZ3YlhObUluMC5UVjRrbkpHbGphWE1wYUhSMGNGOWhTVU5v
- DB_URL: bW9uZ29kK3NydjpzdWJzZG1pbjE6TThtc3czbk41eGZpbmFhcHBuZXRzJTJGf3RydWV8aXBhcG5hbWUlMjNmc3VibmV0JTJGc3VibmV0JmFwcG5hbWU9Y2xzdGVuQHNl
- SECRET: c2h1YmhhbQ==

3. **Update Kubernetes Secret Manifest**: Include the base64 encoded values in your Kubernetes Secret manifest file.



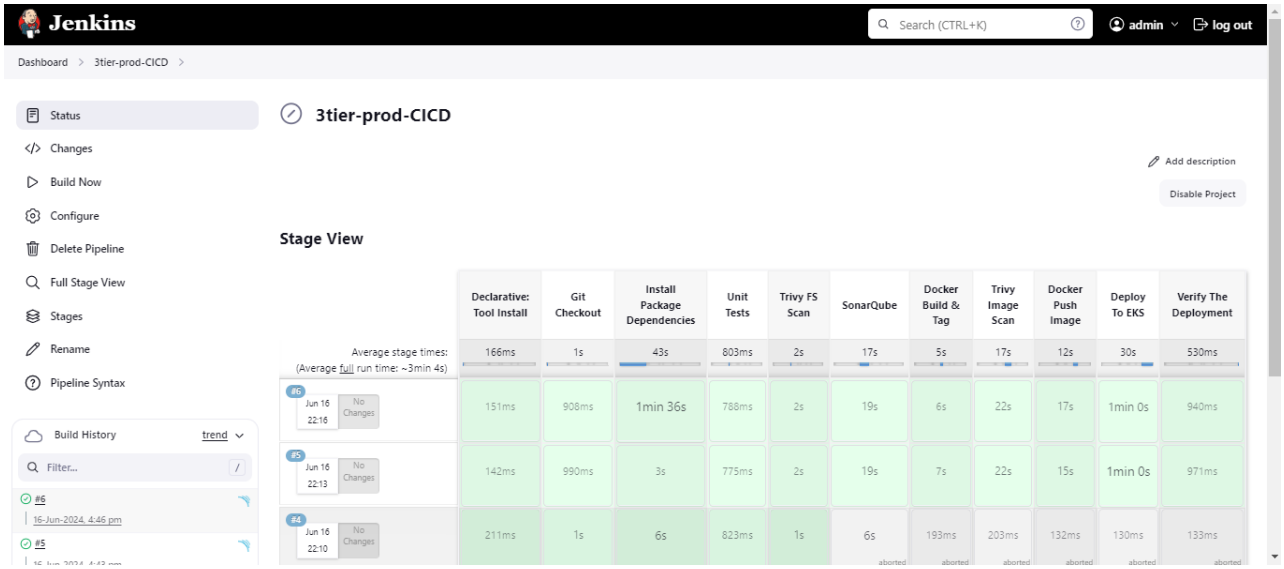
4. **Reference the Secrets in Your Deployment Manifest:** Update your deployment manifest to reference these secrets.

Deployment Pipeline: **Add these to the Jenkins Pipeline**.

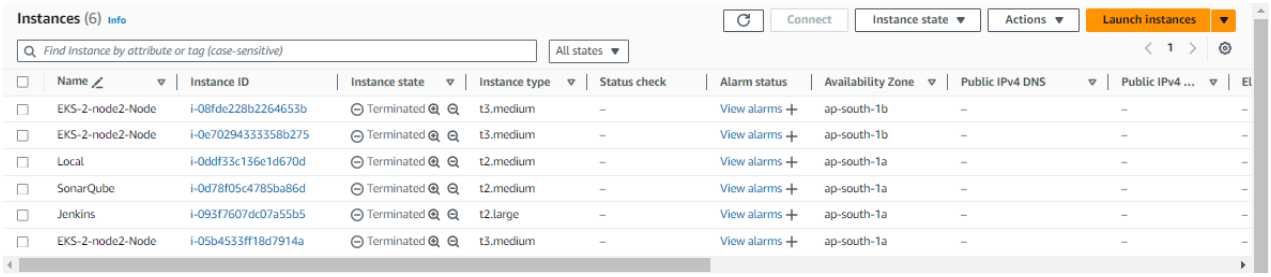
```
   stages {
      stage('Deploy To EKS') {
         steps {
            withKubeCredentials(kubectlCredentials: [[caCertificate: '', clusterName: 'EKS-2', contextName: '', credentialsId: 'k8-token', namespace: 'webapps', serverUrl: 'https://B7C7C20487B2624AAB0AD54DF1469566.yl4.ap-south-1.eks.amazonaws.com']]) {
               sh "kubectl apply -f Manifests/dss.yml"

            }
         }
      }

      stage('verify Deployment') {
         steps {
            withKubeCredentials(kubectlCredentials: [[caCertificate: '', clusterName: 'EKS-2', contextName: '', credentialsId: 'k8-token', namespace: 'webapps', serverUrl: 'https://B7C7C20487B2624AAB0AD54DF1469566.yl4.ap-south-1.eks.amazonaws.com']]) {
               sh "kubectl get svc -n webapps"
               sh "kubectl get pods -n webapps"
            }
         }
      }
   }
}
```

Results

Prod Environment:

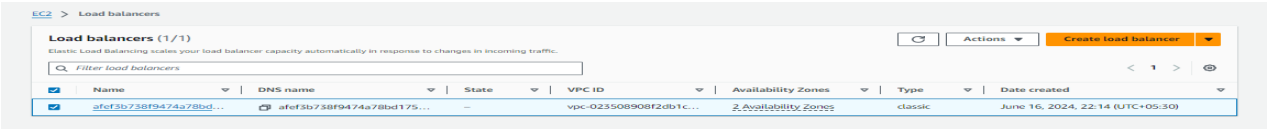


Servers

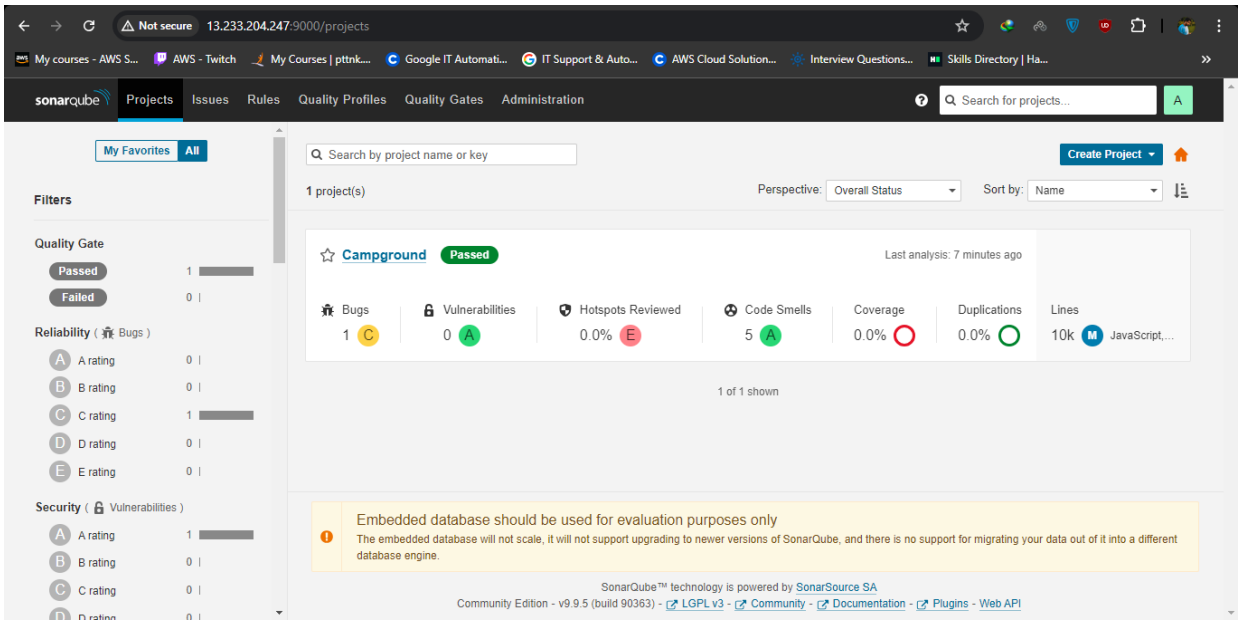


(Forgot to take screenshot during deployment....SORRY)

Load Balancer



SonarQube



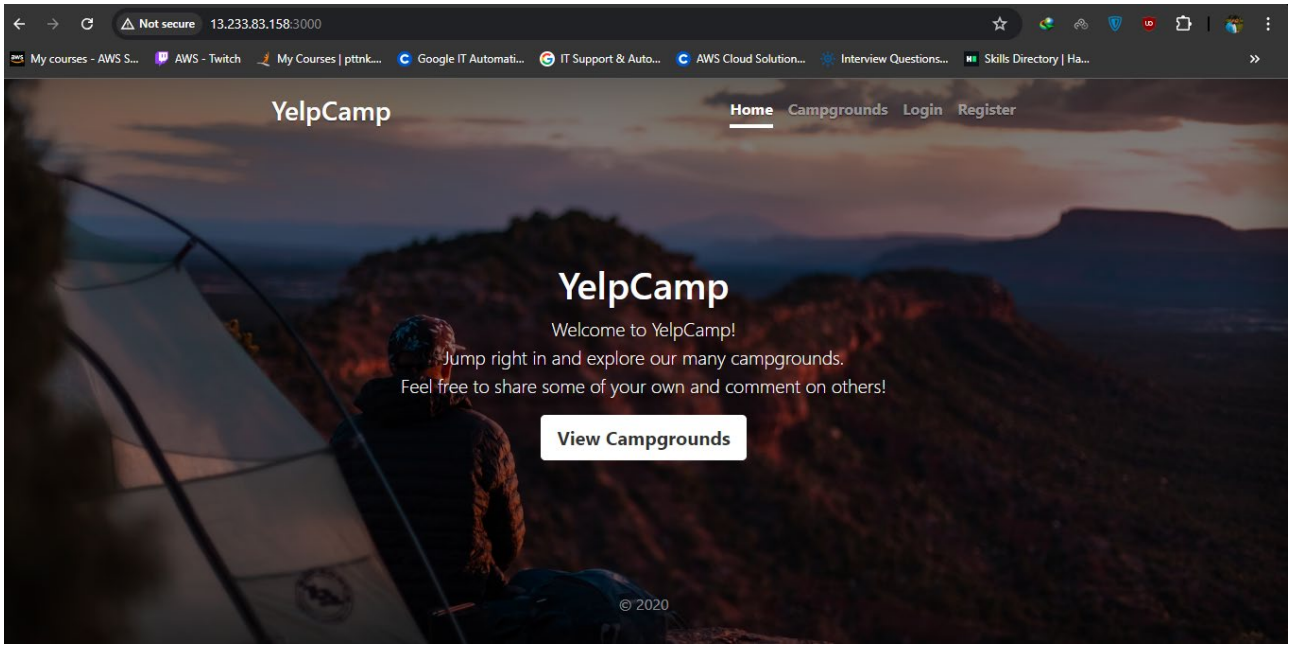
Nodes

```
ubuntu@ip-172-31-32-97:~/script$ kubectl get nodes
NAME                                           STATUS   ROLES    AGE     VERSION
ip-192-168-44-178.ap-south-1.compute.internal   Ready    <none>   3m10s   v1.30.0-eks-036c24b
ip-192-168-51-240.ap-south-1.compute.internal   Ready    <none>   3m12s   v1.30.0-eks-036c24b
ip-192-168-6-57.ap-south-1.compute.internal     Ready    <none>   3m13s   v1.30.0-eks-036c24b
ubuntu@ip-172-31-32-97:~/script$
```

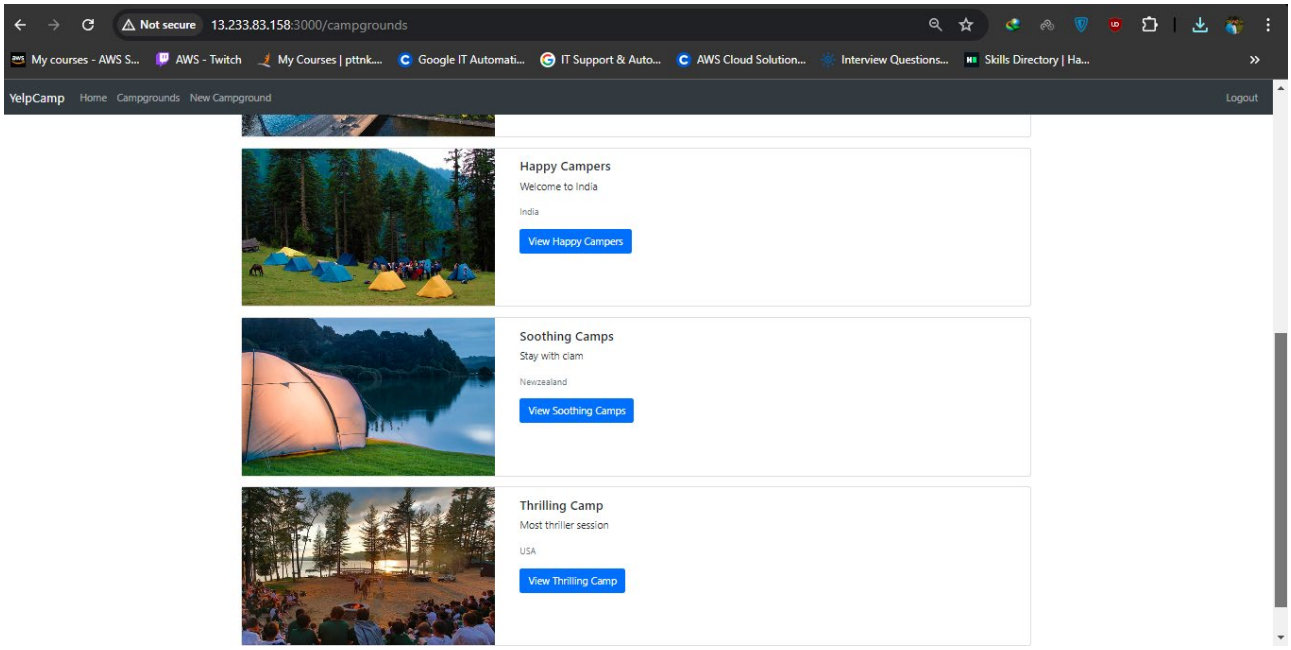
i-093f7607dc07a55b5 (Jenkins)

PublicIPs: 52.66.85.110 PrivateIPs: 172.31.32.97

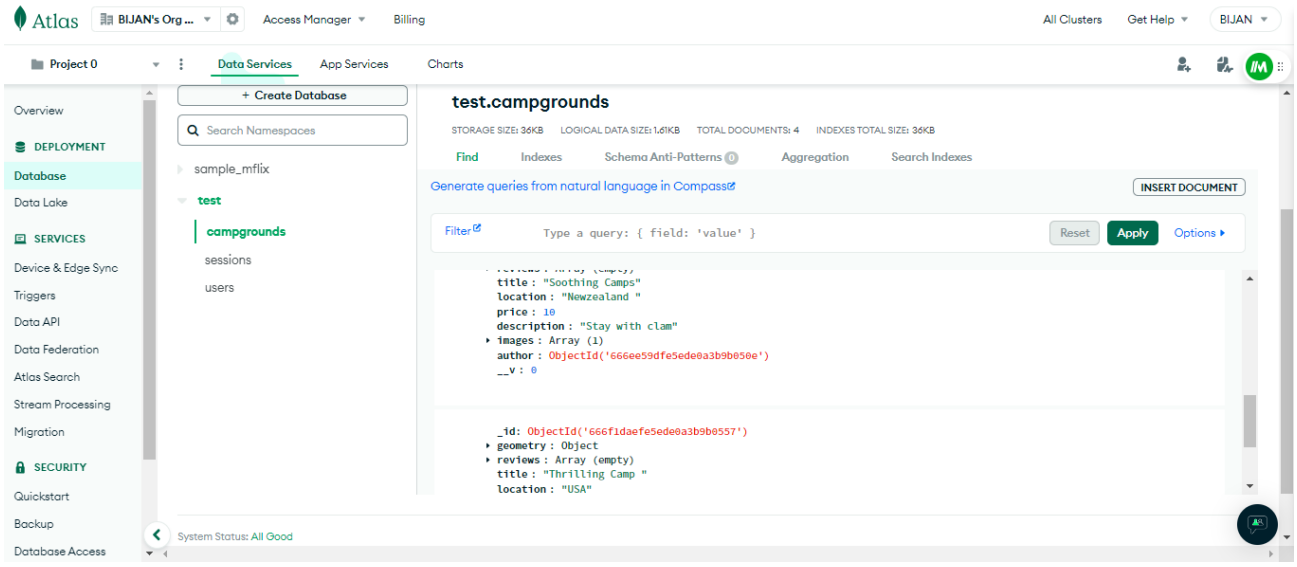
Website



Website



Database



Key Takeaways from the Project

1. **Understanding of Three-Tier Architecture**:
    - Gained practical experience in designing and implementing a three-tier architecture, encompassing the presentation, logic, and data layers.
2. **Proficiency with DevOps Tools**:
    - Improved skills in using key DevOps tools such as Docker, Jenkins, Kubernetes, and SonarQube for building, deploying, and managing applications.
3. **CI/CD Implementation**:
    - Learned how to set up a continuous integration and continuous deployment (CI/CD) pipeline to automate the testing, building, and deployment processes.
4. **Environment Management**:
    - Developed the ability to manage different deployment environments (Local, Development, Production) effectively, ensuring consistency and reliability across stages.
5. **Security Best Practices**:
    - Implemented security best practices by using Kubernetes Secrets for managing sensitive information, ensuring secure handling of environment variables.
6. **Problem-Solving and Troubleshooting**:
    - Enhanced problem-solving and troubleshooting skills by addressing various challenges encountered during the deployment and management of the application.

Acknowledgement

I would like to extend my heartfelt gratitude to **Mr. Aditya Jaiswal** for his invaluable assistance and guidance throughout the course of this project. His comprehensive tutorials and insights on the YouTube channel "**DevOps Shack**" were instrumental in the successful completion of this project. Without his support, this endeavor would not have been possible. I am sincerely thankful for his contributions to my learning and growth in the field of DevOps.

Conclusion

In conclusion, this "**Three-tier Full Stack Project**" has provided an extensive understanding of how to build, deploy, and manage applications across different environments - Local, Development, and Production. By leveraging modern DevOps tools and practices, such as Docker, Jenkins, Kubernetes, and SonarQube, this project has showcased a robust workflow for continuous integration and continuous deployment (CI/CD). The structured approach to environment management and the implementation of security best practices highlight the importance of maintaining application integrity and performance across all stages of development. This project not only reinforces technical skills but also emphasizes the significance of meticulous planning and execution in achieving seamless application delivery.