Deploy Golang Two Tier Go Application on Kubernetes with CI/CD Pipeline

Author: **Nitesh Nepali**

DevOps Engineer/ Cyber Security Enthusiast: **(CCNA, AWS-SAA, Linux (RHCSA), DevSecOps)**

Email: pingnitesh07@gmail.com

Project Overview

Assessment:

Set up a CI/CD pipeline for a **Go** OR Java **Spring Boot** application using any CI/CD tools and containerize the application. The application must be deployed in a VM (you may use EC2 or GCP VM instance) using Kubernetes and helm. Implement monitoring to monitor your resource consumption.

Requirements:

1. **The entire system must be built using Terraform for dev and prod environments**.
2. Optimize resource utilization within the Kubernetes cluster. **Implement Horizontal Pod Autoscaling (HPA) based on CPU or custom metrics.**
3. The docker image must be optimized.
4. The application must be highly available and secure and optimize cost as much as possible.
5. Manage container replicas, and ensure high availability and scalability. **Configure Persistent Volume (PV) and Persistent Volume Claims (PVC).**
6. Configure health checks.

Some boilerplates for reference:

```
wget https://autostrada.dev/download/7u7gfr6nxhbai.zip
unzip 7u7gfr6nxhbai.zip
https://github.com/spring-projects/spring-boot
```

Tools and Services Used

This project "**Two Tier Go Application deploy on Kubernetes cluster** " involves a complete cloud-native setup using various AWS services, DevOps tools, and third-party platforms. Below is a categorized list of the tools and services used:

AWS Services

1. Amazon VPC – For custom network configuration with public and private subnets
2. Amazon EC2 – To launch Ubuntu instances and host intermediate setup and for Jenkins
3. IAM (Identity and Access Management) – For creating users and assigning permissions
6. Auto Scaling Group (ASG) – To scale EC2 instances automatically
7. Elastic Load Balancer (ELB) – To distribute traffic and to Kubernetes cluster
8. Amazon Route 53 – For domain name system (DNS) configuration

Infrastructure as a Code tool: terraform and terraform vault

Kubernetes cluster setup: Kops and terraform

Kubernetes package manager: helm

Containerization tool: docker and docker hub

Container orchestration: Kubernetes

CI/CD tool: Jenkins and ArgoCD

Monitoring and Alerting: Prometheus and Grafana

Code Repository and Version control System: Git and GitHub

Local Development & Testing Tools

1. **Git** – For cloning and version control
2. **Docker** – To containerize the application
3. **AWS CLI** – To interact with AWS services from the terminal

Third-Party Tools

1. **Mercantile domain registration** – For domain registration and DNS management
2. **Web-based Name Server Checker** – To verify Route 53 propagation

Phase 1: Initial Setup and Local Development

Step 1: Launch EC2 (Ubuntu 22.04):

- Provision an EC2 instance on AWS with Ubuntu 22.04.
- Connect to the instance using SSH.

Step 2: Clone the Code:

- Update all the packages and then clone the code.
- Clone your application's code repository onto the EC2 instance.

```
sudo apt update && upgrade -y
sudo apt install git -y
sudo git clone https://github.com/Nitesh0ne/Golang_codebase.git
```

Step 3: Install Golang, PostGreSql and Setup Database

A. **Install golang and PostGreSql**
   - sudo install golang -y
   - sudo apt install postgresql postgresql-contrib -y

B. **Setup Database:**
   a. Switch to the postgres user:
      - sudo -i -u postgres
   b. Access the PostgreSQL Shell
      - psql
   c. Create a New User (Role)
      - CREATE USER myuser WITH PASSWORD 'mypassword';
   d. Create a New Database
      - CREATE DATABASE mydb OWNER myuser;
   e. Create a New User (Role)
      - ALTER DATABASE mydb OWNER TO myuser;
   f. Grant Privileges to the User
      - GRANT ALL PRIVILEGES ON DATABASE mydb TO myuser;
   g. Exit PostgreSQL and postgres User.
      - \q
      - Exit

C. **Export DB String as environment variable:**

- export DB_DSN="postgres://myuser:mypassword@localhost:5432/db?sslmode=disable"

D. Build Golang code and run the app:

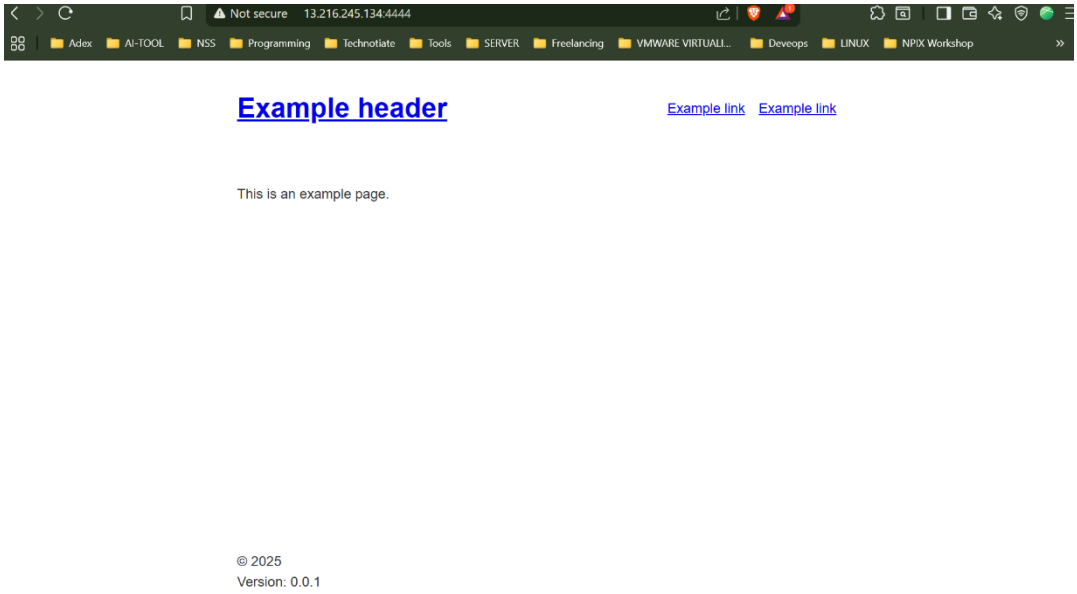
a. **Navigate to the clone repo and download the dependencies**

- cd Golang_codebase
- go mod tidy

b. **Navigate to folder locating main.go**

- cd ~/Golang_codebase/cmd/web
- go run .

E. Test the Connection on port 4444
http://<ubuntu -IP>:<4444> you will see the page shown in below



Phase 2: Infrastructure Setup

In this phase, Kubernetes cluster for dev and prod is provision using Kops and Terraform.

KOPS Configuration Procedures

Step 1: Launch EC2 (Ubuntu 22.04) as Management server:

- Provision an EC2 instance on AWS with Ubuntu 22.04.
- Connect to the instance using SSH.
- Management server is used for managing Kubernetes cluster.

Step 2: Install Kubectl

- curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
- curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.9.0/bin/linux/amd64/kubectl
- chmod +x ./kubectl
- sudo mv ./kubectl /usr/local/bin/kubectl
- export PATH=$PATH:/usr/local/bin/
- kubectl

  Note: run all command as root user

Step 2 - Install kops

  #Install "kops" on Linux

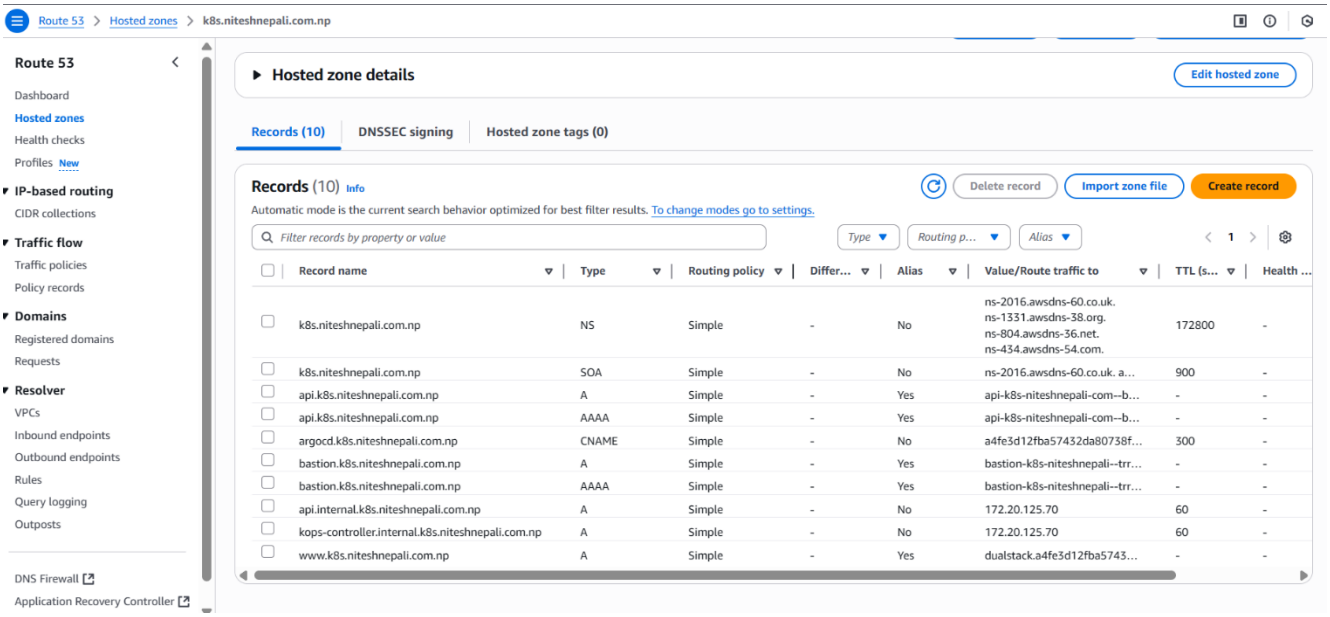
- yum install wget
- curl -LO https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag_name | cut -d '"' -f 4)/kops-linux-amd64
- chmod +x kops-linux-amd64
- sudo mv kops-linux-amd64 /usr/local/bin/kops

- kops upgrade
  Reference - https://github.com/kubernetes/kops

Step 3 - Configure AWS Route 53

DNS Changes (Refer the images show below;)

- Create a public Hosted Zone in Route53 k8s.niteshnepali.com.np
- Get a NS & SOA
- Go to Your domain provider and update the Name Server.
- Enter the NS records



Step 4 - Create a IAM Role or IAM User and Assign access to management Server

Create a IAM user or IAM Role with following Permission.

- AmazonEC2FullAccess
- AmazonRoute53FullAccess
- AmazonS3FullAccess
- IAMFullAccess
- AmazonVPCFullAccess
- AmazonSQSFullAccess

- AmazonEventBridgeFullAccess

In Case of IAM Role attach to the management server
In Case of IAM user configure access key and secret key

Step 6 - Craete a S3 bucket

```
# Create and then list a new S3 bucket
$ aws s3 mb s3://k8s.niteshnepali.com.np
$ aws s3 ls
```

Step 7 – Generating a terraform configuration using Kops

```
kops create cluster \
--name=k8s.niteshnepali.com.np \
--state=s3://k8s.niteshnepali.com.np \
--zones=ap-south-1a, ap-south-1b \
--node-count=2 \
--control-plane-count=1 \
--node-size=t3.medium \
--control-plane-size=t3.medium \
--control-plane-zones=us-east-1a \
--control-plane-volume-size=30 \
--node-volume-size=30 \
--topology=private \
--networking=calico \
--bastion \
--ssh-public-key=~/.ssh/id_rsa.pub \ #ssh public key path
--dns-zone=k8s.niteshnepali.com.np \
--target=terraform \
--out=terraform \
--yes
```

Note: for remote access of the bastion host generate the ssh public key and provide the path

Step 8 – Creating Cluster

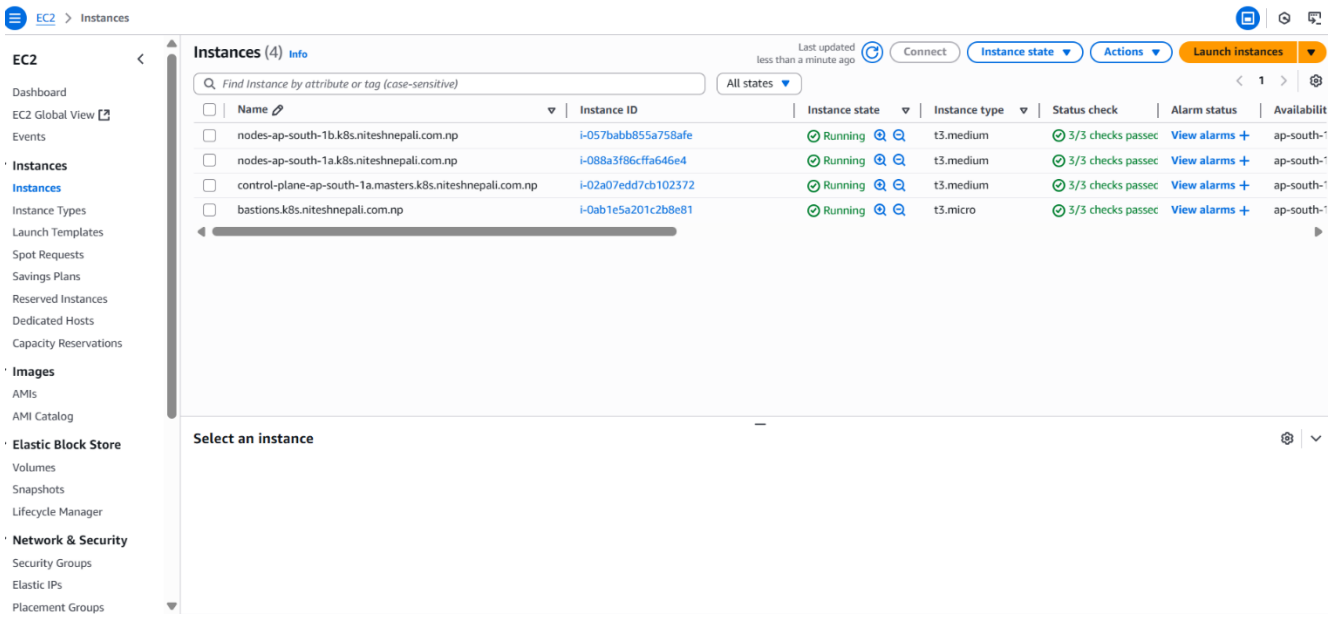
First install the terraform using official documentation.

- **cd terraform**
- **terraform init**
- **terraform plan**
- **terraform apply**

Step 9 – Updating the Cluster

- kops get cluster --name k8s.niteshnepali.com.np -o yaml > cluster.yaml
- kops replace -f cluster.yaml --state=s3://niteshnepali.com.np
- kops update cluster --name k8s.niteshnepali.com.np --state=s3://k8.niteshnepali.com.np --out=./terraform --target=terraform

then repeat Step no 8



Phase 3: Dockerizing the app and setup CI/CD

Dockerizing the golang code

Step 1. Install docker on management server

sudo apt install docker.io

Step 2. Clone the Repo

git clone https://github.com/Nitesh0ne/Golang_codebase.git

Step 3. Create a DockerFile

```
# Build Stage
FROM golang as builder
#set environment variables
ENV CGO_ENABLED=0 GOOS=linux

WORKDIR /opt

# Copy source files
COPY . .

#download dependencies
RUN go mod tidy

# Build the Go application
RUN go build -v -o app  ./cmd/web

#Final Stage
FROM debian:bullseye-slim

# Create non-root user and group
RUN groupadd -r goapp && useradd -r -g goapp goapp

WORKDIR /app

COPY --from=builder /opt/app /usr/local/bin/app

# Set permissions
RUN chown goapp:goapp /usr/local/bin/app

# Switch to non-root user
USER goapp

# Expose the port the app listens on
EXPOSE 4444

# Run the app
CMD ["app"]
```

Step 4. Install Jenkins and setup pipeline

- sudo wget -O /etc/apt/keyrings/jenkins-keyring.asc \
  https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
  echo "deb [signed-by=/etc/apt/keyrings/jenkins-keyring.asc]" \
  https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
  /etc/apt/sources.list.d/jenkins.list > /dev/null
- sudo apt-get update
- sudo apt-get install Jenkins

  Note: For the Jenkins installation and java compatible follow Jenkins official documentation

Step 5. Crete a Jenkin file and run the pipeline

First, create a docker hub credentials and configure on Jenkins to use it

```
pipeline{
    agent {label 'linux' }

    tools{
        git 'Default'
    }

    environment {
        IMAGE_NAME = 'nitace/go-lang-app'
        IMAGE_TAG  = "v${env.BUILD_NUMBER}"
        REGISTRY_CREDENTIALS = 'docker-hub-credentials'
        }
    stages{


        stage ('Build Image'){
            steps{
               sh 'docker build -t ${IMAGE_NAME}:${IMAGE_TAG} .'
            }
        }

        stage ('PUSH TO REGESTRY'){
           steps {
                withCredentials([usernamePassword(credentialsId: "${env.REGISTRY_CREDENTIALS}", usernameVariable: 'USERNAME', passwordVariable: 'PASSWORD')]) {
            script {
                sh """
                    echo "$PASSWORD" | docker login -u "$USERNAME" --password-stdin
                    docker push ${IMAGE_NAME}:${IMAGE_TAG}
                """
                    }
                }
            }
        }
    }
}
```

Phase 3: Monitoring (Prometheus and Grafana)

We are using helm package manager to install Prometheus and Grafana.

Step 1. Install helm on the Management Server

- $ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
- $ chmod 700 get_helm.sh
- $ ./get_helm.sh

Step 2. Add Prometheus Repo and update it and installed it

- helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
- helm repo update
- helm install prometheus prometheus-community/kube-prometheus-stack --namespace monitoring --create-namespace

Get Grafana 'admin' user password by running:

kubectl --namespace monitoring get secrets prometheus-grafana -o jsonpath="{.data.admin-password}" | base64 -d ; echo

Step 3. Exposing service via ingress

- helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
- helm repo update
- helm install ingress-nginx ingress-nginx/ingress-nginx --namespace ingress-nginx --create-namespace

Step 4. Create ingress file for both Prometheus and Grafana

For Grafana

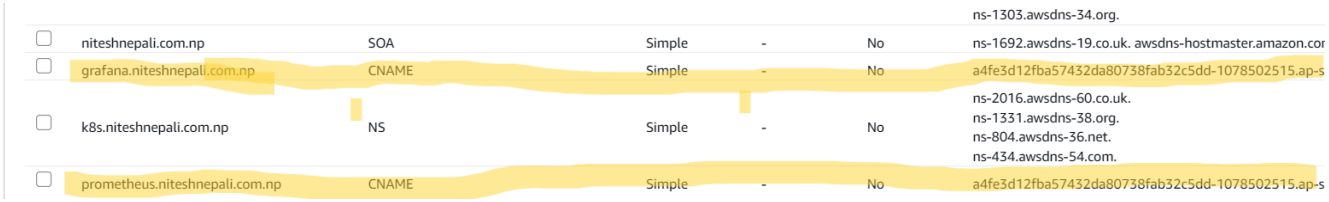
create grafana_ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: grafana-ingress
  namespace: monitoring
  annotations:
    kubernetes.io/ingress.className: "nginx"                    # or "alb" if using AWS ALB ingressI
    nginx.ingress.kubernetes.io/rewrite-target: /
    # Optional: enable basic auth or TLS here
spec:
  rules:
  - host: grafana.niteshnepali.com.np
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: prometheus-grafana
            port:
              number: 80
```

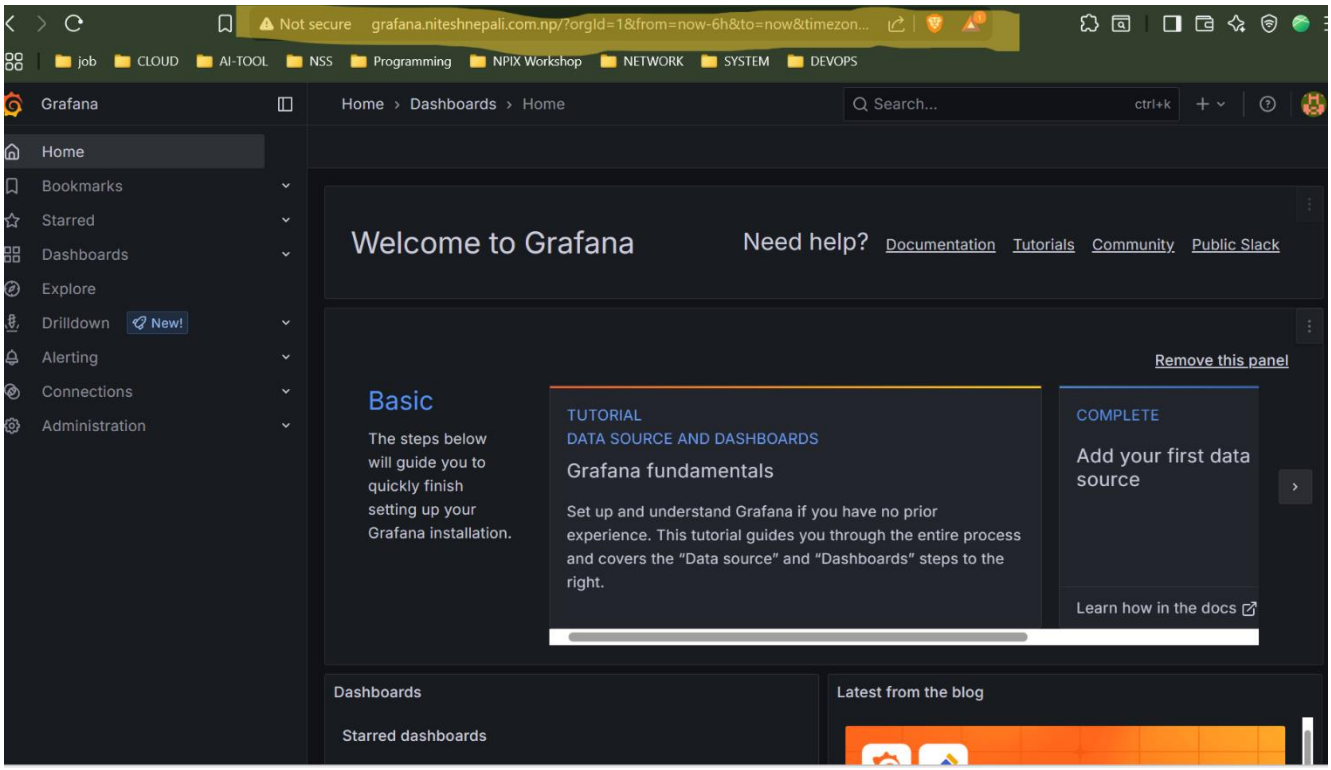
For Prometheues
Create prometheues_ingress.yaml

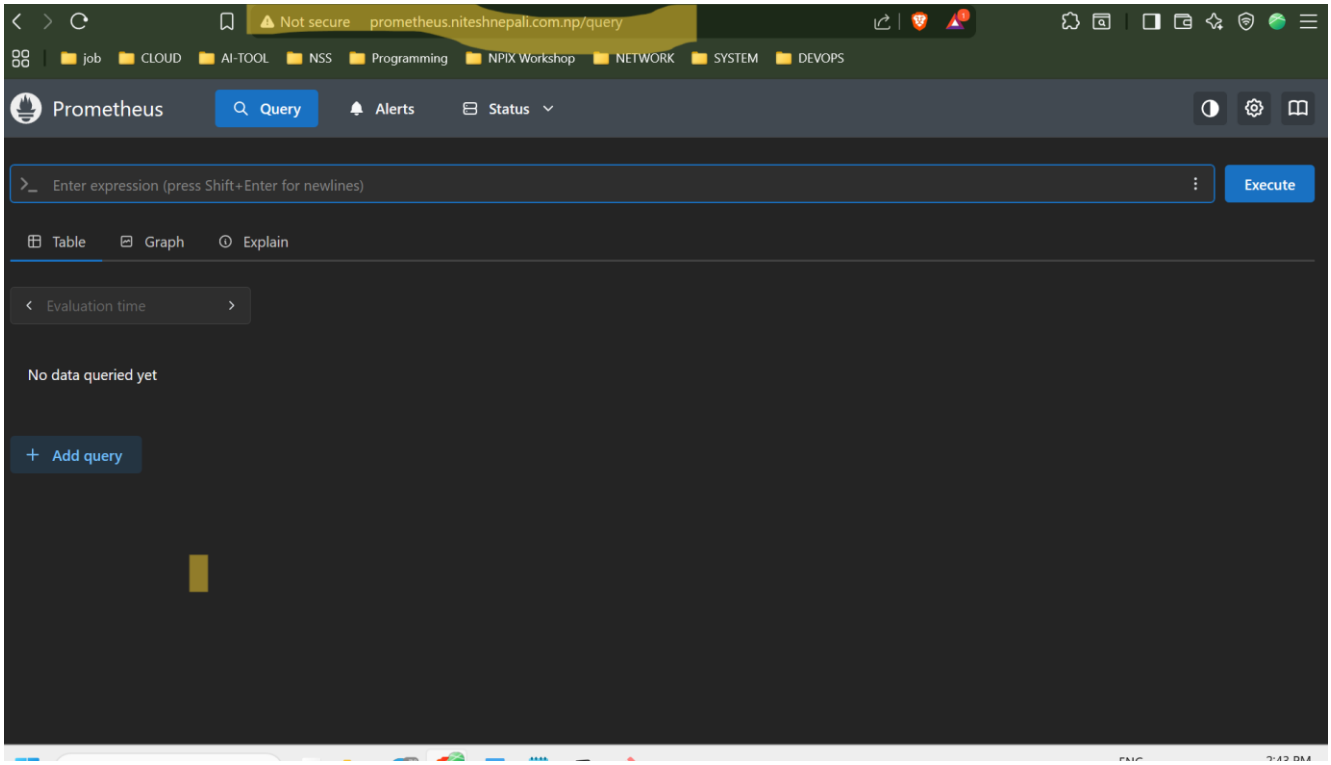
```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: prometheus-ingress
  namespace: monitoring
  annotations: {}
spec:
  ingressClassName: nginx
  rules:
  - host: prometheus.niteshnepali.com.np
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: prometheus-kube-prometheus-prometheus
            port:
              number: 9090
```

Step 5. Create a CNAME record in Route53 and point to ingress controller and access it.



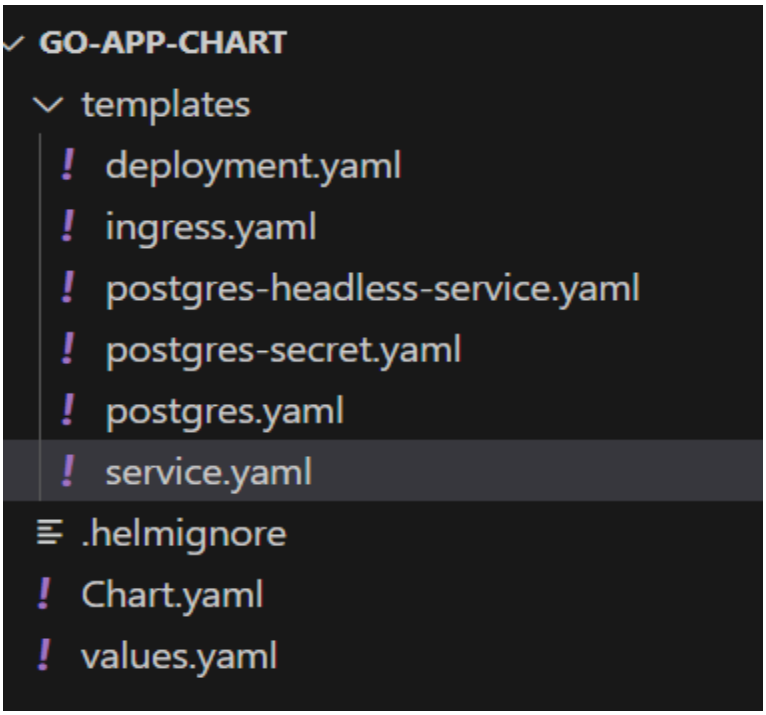
						ns-1303.awsdns-34.org.
☐	niteshnepali.com.np	SOA	Simple	-	No	ns-1692.awsdns-19.co.uk. awsdns-hostmaster.amazon.cor
☐	grafana.niteshnepali.com.np	CNAME	Simple	-	No	a4fe3d12fba57432da80738fab32c5dd-1078502515.ap-s
☐	k8s.niteshnepali.com.np	NS	Simple	-	No	ns-2016.awsdns-60.co.uk. ns-1331.awsdns-38.org. ns-804.awsdns-36.net. ns-434.awsdns-54.com.
☐	prometheus.niteshnepali.com.np	CNAME	Simple	-	No	a4fe3d12fba57432da80738fab32c5dd-1078502515.ap-s


Not secure grafana.niteshnepali.com.np/?orgId=1&from=now-6h&to=now&timezon...
job CLOUD AI-TOOL NSS Programming NPIX Workshop NETWORK SYSTEM DEVOPS
Grafana
Home › Dashboards › Home
Search... ctrl+k
Home
Bookmarks
Starred
Dashboards
Explore
Drilldown New!
Alerting
Connections
Administration
Welcome to Grafana
Need help? Documentation Tutorials Community Public Slack
Remove this panel
Basic
The steps below will guide you to quickly finish setting up your Grafana installation.
TUTORIAL
DATA SOURCE AND DASHBOARDS
Grafana fundamentals
Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.
COMPLETE
Add your first data source
Learn how in the docs
Dashboards
Starred dashboards
Latest from the blog


Not secure prometheus.niteshnepali.com.np/query
job CLOUD AI-TOOL NSS Programming NPIX Workshop NETWORK SYSTEM DEVOPS
Prometheus
Query
Alerts
Status
Enter expression (press Shift+Enter for newlines)
Execute
Table
Graph
Explain
Evaluation time
No data queried yet
Add query

Phase 5. Creating helm chart for Kubernetes Deployment

Creating a helm chart Folder Structure



template/deployment.yaml

```yaml
# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-deploy
  labels:
    env: production
    tier: backend
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      env: production
      tier: backend
  template:
    metadata:
      labels:
        env: production
        tier: backend
    spec:
      initContainers:
        - name: wait-for-postgres
          image: busybox
          command:
            - sh
            - -c
            - |
              until nc -z postgres-0.postgres-headless.default.svc.cluster.local 5432; do
                echo "Waiting for Postgres TCP connection..."
                sleep 3
              done
      dnsPolicy: ClusterFirst
      containers:
        - name: testcont
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          ports:
            - containerPort: 4444
          env:
            - name: DB_DSN
              valueFrom:
                secretKeyRef:
                  name: postgres-secret
                  key: DB_DSN
          readinessProbe:
            tcpSocket:
              port: 4444
            initialDelaySeconds: 5
            periodSeconds: 10
            failureThreshold: 3
          livenessProbe:
            tcpSocket:
              port: 4444
            initialDelaySeconds: 15
            periodSeconds: 20
            failureThreshold: 3
```

template/service.yaml

```yaml
# templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-svc
  labels:
    env: production
    tier: backend
spec:
  type: {{ .Values.service.type }}
  selector:
    env: production
    tier: backend
  ports:
    - protocol: TCP
      port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.targetPort }}
```

Template/postgres.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: {{ .Values.postgres.name }}
  labels:
    app: {{ .Values.postgres.name }}
spec:
  serviceName: "{{ .Values.postgres.serviceName }}"
  replicas: {{ .Values.postgres.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Values.postgres.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.postgres.name }}
    spec:
      containers:
        - name: {{ .Values.postgres.name }}
          image: "{{ .Values.postgres.image }}"
          imagePullPolicy: IfNotPresent
          envFrom:
            - secretRef:
                name: {{ .Values.postgres.secretName }}
          ports:
            - containerPort: 5432
          volumeMounts:
            - name: postgres-data
              mountPath: /var/lib/postgresql/data3
  volumeClaimTemplates:
    - metadata:
        name: postgres-data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: {{ .Values.postgres.storage }}
```

postgres-headless-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.postgres.serviceName }}
spec:
  clusterIP: None
  selector:
    app: {{ .Values.postgres.name }}
  ports:
    - port: 5432
      name: db
```

postgres-secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: {{ .Values.postgres.secretName }}
type: Opaque
stringData:
  POSTGRES_USER: {{ .Values.postgres.auth.user | quote }}
  POSTGRES_PASSWORD: {{ .Values.postgres.auth.password |
quote }}
  POSTGRES_DB: {{ .Values.postgres.auth.database | quote }}
  DB_DSN: "user={{ .Values.postgres.auth.user }} password={{
.Values.postgres.auth.password }} host={{
.Values.postgres.dsnHost }} port=5432 dbname={{
.Values.postgres.auth.database }} sslmode=disable"
```

Ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ .Release.Name }}-ingress
  labels:
    app: {{ .Release.Name }}
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  ingressClassName: nginx
  rules:
    - host: {{ .Values.ingress.host | default "example.com"
}}
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: {{ .Release.Name }}-svc
                port:
                  number: {{ .Values.service.port | default
80 }}
```

Phase 6: Setup Argocd and deployment

Step 1. Create Namespace for ArgoCD

- kubectl create namespace argocd

Step 2. Add ArgoCD Helm Repository

- helm repo add argo https://argoproj.github.io/argo-helm
- helm repo update

Step 3. Install ArgoCD via Helm

- helm install argocd argo/argo-cd \
  --namespace argocd

Step 4. Expose argocd via ingress

Argocd-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: argocd-ingress
  namespace: argocd
  annotations:
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
spec:
  ingressClassName: nginx
  rules:
  - host: argocd.k8s.niteshnepali.com.np
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: argocd-server
            port:
              number: 443
```

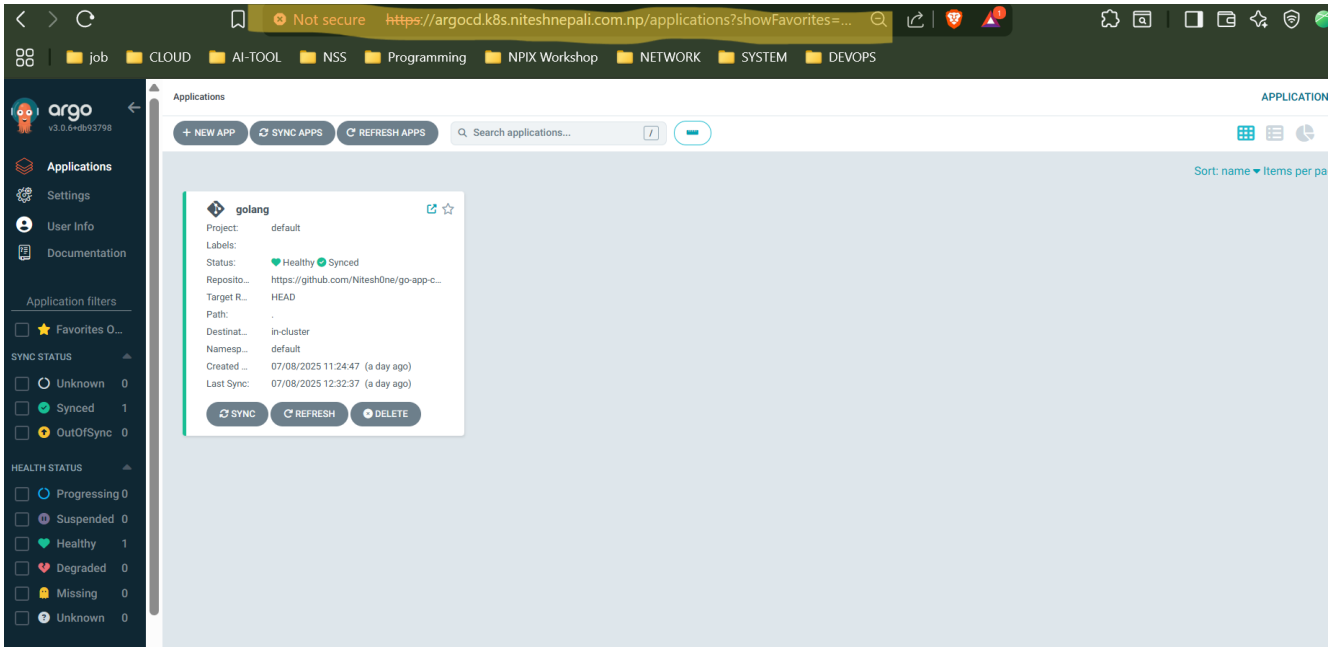
Step 5. Create Route53 CNAME Record and point to ingress controller

api.k8s.niteshnepali.com.np	AAAA	Simple	-	Yes	api-k8s-niteshnepali-com--b503ee-4e03a8b6c0d0f619.elb.ap-sout...
argocd.k8s.niteshnepali.com.np	CNAME	Simple	-	No	a4fe3d12fba57432da80738fab32c5dd-1078502515.ap-south-1.elb...

Step 6: Acess argocd

Get argocd credentials running command

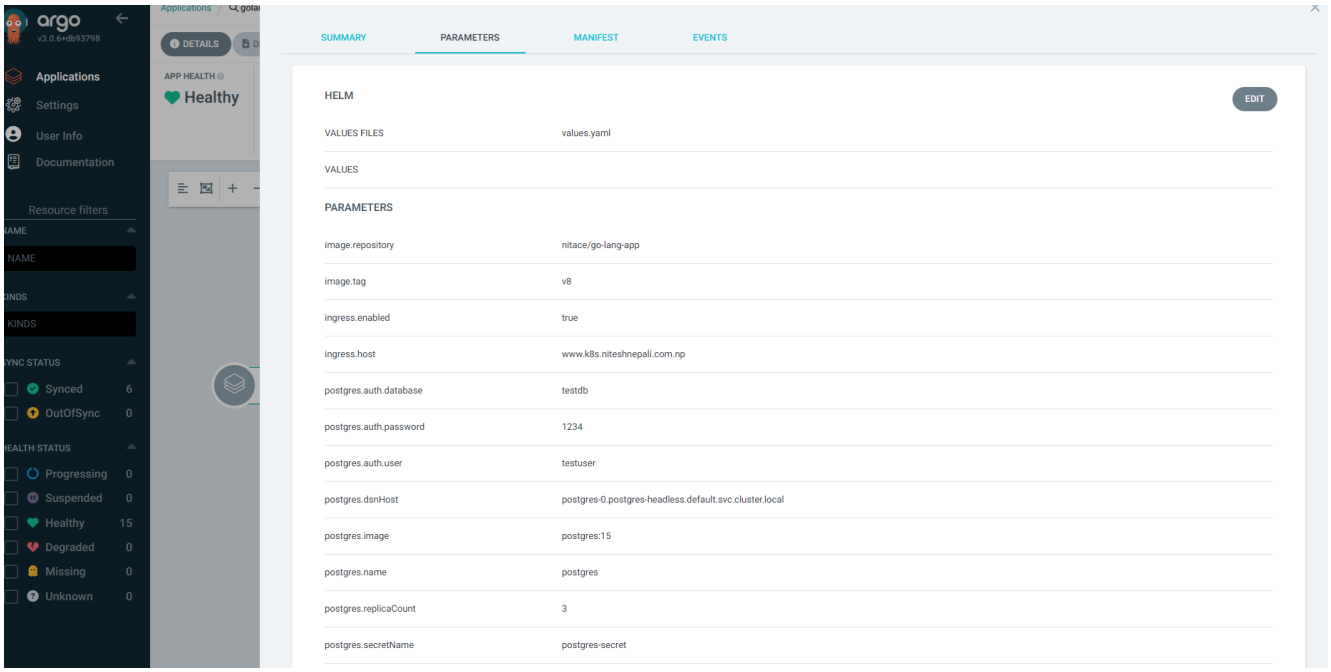
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d



Phase 7: Deployment using argocd

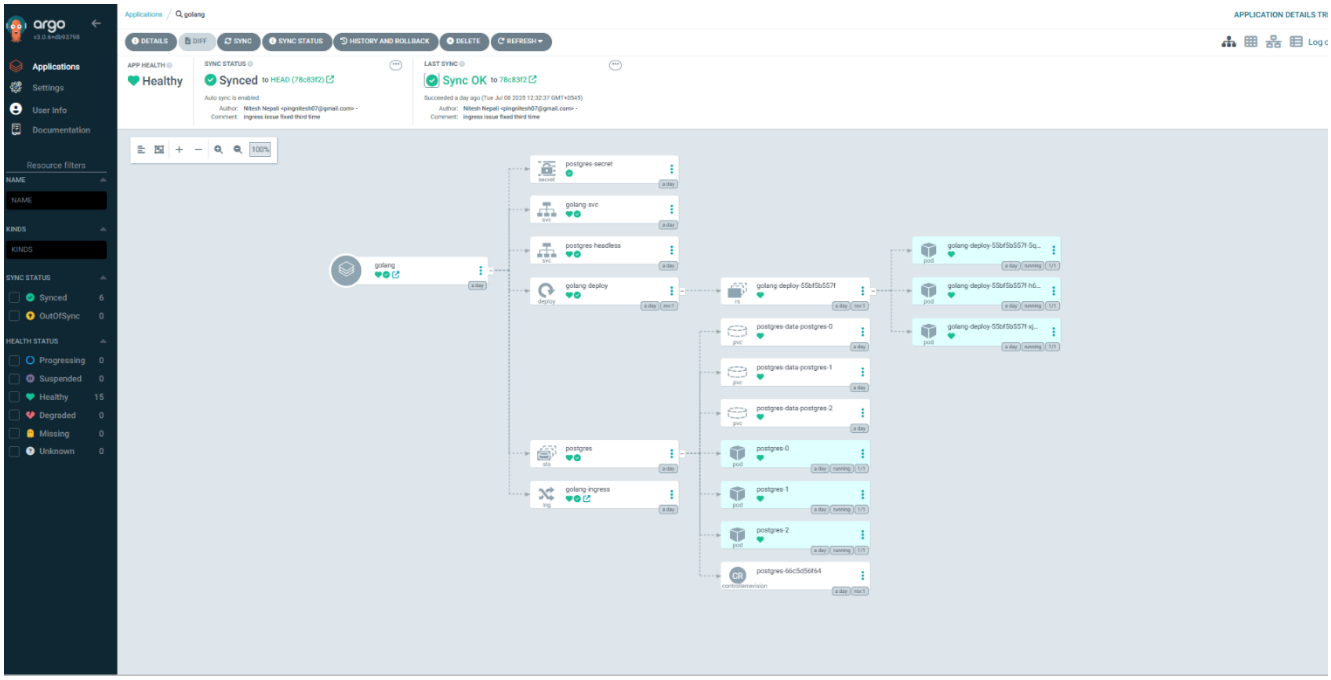
Step 1. Create a helm repo and push to the github

Step 2. Confiure the argocd for deployment



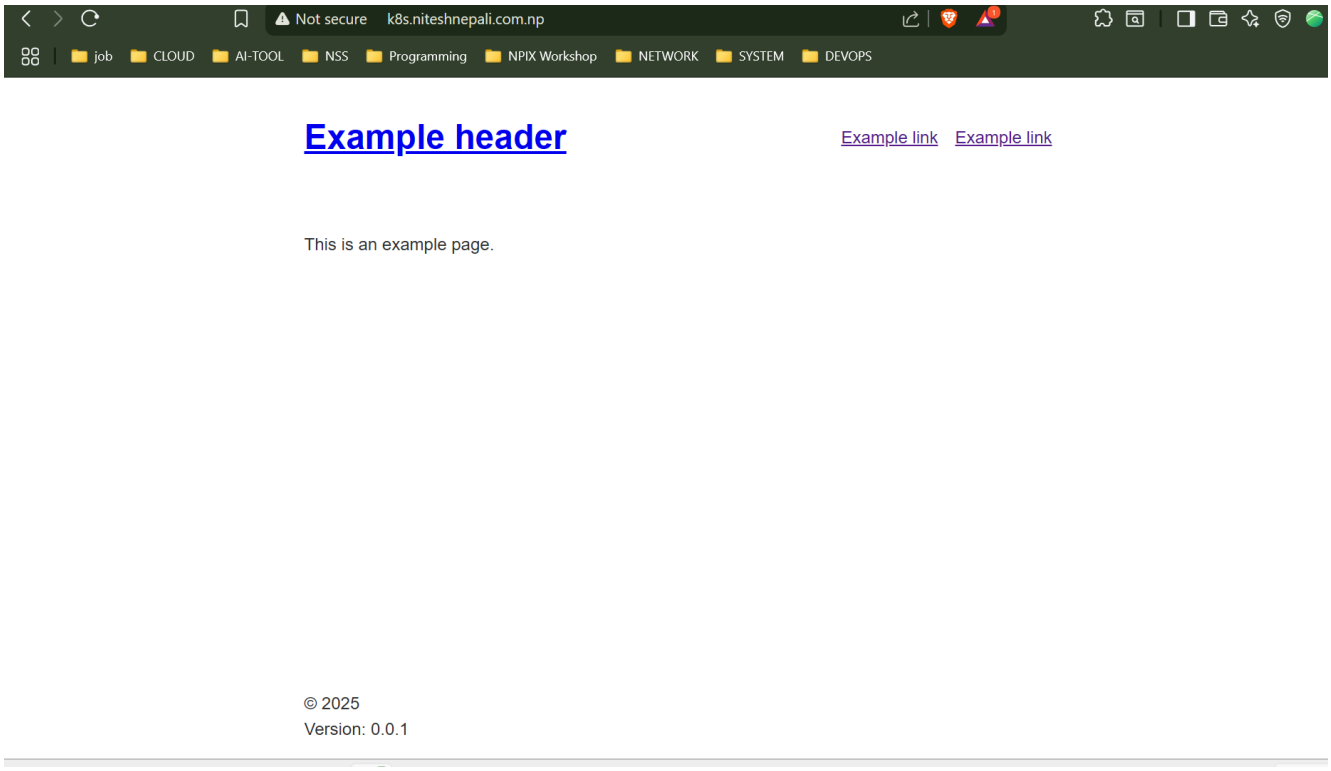
Step 6. Verify the deployment

3 pod of the golang conatainer
3 pod of the postgres database
Ingress controller for the application,



Step 6: create the Route53 CNAME record for the application and access it

Step7. Access the application



Project Repo:

Kubernetes cluster using kops and Terrform repo:
https://github.com/Nitesh0ne/kops_cluster_infra_setup.git
Go Application with Dockerfile and Jenkins File:
https://github.com/Nitesh0ne/Golang_codebase.git
Kubernetes helm chart repo:
https://github.com/Nitesh0ne/go-app-chart.git