Banque Misr internship 2024 Graduation Project

Library Management System

Team members (Group A):

Rana Essam

Gamila Amr

Basma Ashraf

Farah El-Ashqar

Project Objective:

The objective of this DevOps final project is to comprehensively integrate application development, Dockerization, Kubernetes deployment, CI/CD automation, and infrastructure management with Terraform. This project's main goal is to create a well operating Library Management System using the mentioned steps above.

Frontend

React: A JavaScript library for building user interfaces.

Node.js: Used for running the React development server.

Backend

Flask: A lightweight WSGI web application framework in Python.

Python: The programming language used for the backend logic.

Project Overview:

The project is divided into the following parts:

1.Application Development

2.Dockerization

3.Infrastructure as Code with Terraform

4.Kubernetes Deployment on EKS

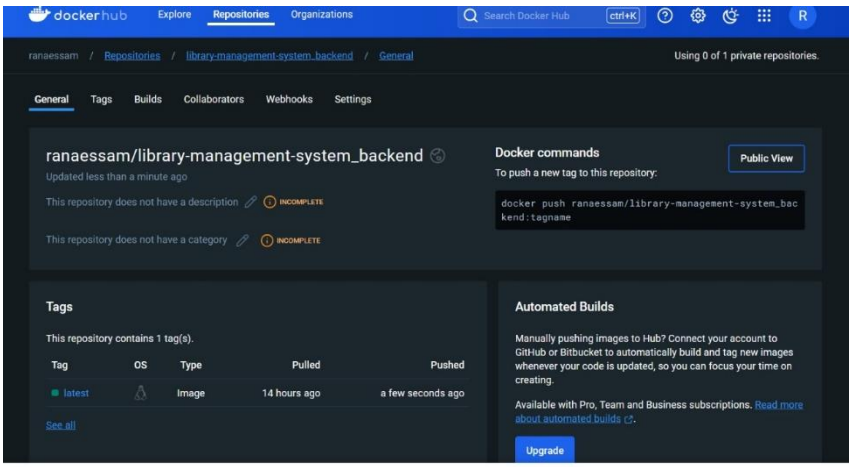
5.CI/CD Pipeline Setup

1.Application Development:

In this phase, we set up a Python virtual environment, installed Flask, and created a simple Flask application to serve as the Library Management System. This application includes basic routes to get a list of books, retrieve a specific book, and add a new book to the library. By testing the application locally, we ensure that the foundation of the Library Management System is properly set up and functional.

2.Dockerization:

we created a Dockerfile that contains the instructions for building a Docker image. By specifying the base image, setting the working directory, copying the application code, installing dependencies, exposing the necessary port, and defining the command to run the application, we packaged the application into a container. After building and testing the Docker image locally, we pushed it to Docker Hub for sharing and future deployment. This ensures that your application can run consistently across different environments, simplifying the deployment process.



Containerization:

Docker: Used to containerize both the frontend and backend applications.

Docker Compose: Used to define and run multi-container Docker applications.

The docker-compose.yml file defines the services required for this project:

```yaml
version: "3.8"

services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    volumes:
      - ./backend:/app
    environment:
      - FLASK_ENV=development

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "5173:5173"
    volumes:
      - ./frontend:/app
    environment:
      - NODE_ENV=development
```

3.Infrastructure as Code with Terraform:

we used Terraform to define the necessary infrastructure on AWS for deploying the project. The infrastructure includes a Virtual Private Cloud (VPC), subnets, and an Amazon EKS cluster with associated resources. By writing Terraform configuration files, we ensured that the infrastructure setup is repeatable and automated. Once the infrastructure is provisioned, you configure kubectl to interact with the EKS cluster, preparing it for application deployment.

The configuration includes the following components:

***Terraform Backend Configuration:**

Stores the Terraform state file in an S3 bucket with state locking using a DynamoDB table.

***Providers:**

AWS provider configured for the us-east-1 region.

Kubernetes provider to manage Kubernetes resources within the created EKS cluster.

***VPC Configuration:**

A VPC with DNS support and hostnames enabled.

Public and private subnets in different availability zones.

An Internet Gateway for the public subnet.

NAT Gateway for outbound internet access from private subnets.

Route tables for managing network traffic.

***Security Groups:**

Allows inbound traffic on ports 80 (HTTP), 443 (HTTPS), and 5173.

Allows all outbound traffic.

***IAM Roles and Policies:**

IAM role for the EKS cluster with the AmazonEKSClusterPolicy.

IAM role for EKS Node Groups with necessary policies for worker nodes.

IAM role for master access with a custom policy to allow specific users to assume the role.

***EKS Cluster and Node Group:**

EKS cluster configured with the specified VPC and subnets.

Node group with scaling configuration for the number of worker nodes.

***IAM Policies and Role for Master Access:**

Custom IAM policy and role for managing access to the EKS cluster by trusted accounts and users.

Configuration Details:

Terraform Backend:

Bucket: teamm01

Key: terraform.tfstate

Region: us-east-1

DynamoDB Table: terraform-locks

AWS VPC:

VPC CIDR: 10.0.0.0/16

Public Subnet CIDR: 10.0.1.0/24 (Availability Zone: us-east-1a)

Private Subnet CIDR: 10.0.2.0/24 (Availability Zone: us-east-1b)

EKS Cluster:

Cluster Name: Team1-cluster

Node Group Name: Team1-node-group

Desired Node Count: 1

Maximum Node Count: 3

Minimum Node Count: 1

IAM User ARNs Allowed to Assume the Role:

arn:aws:iam::637423483309:user/basma

arn:aws:iam::637423483309:user/gamila

arn:aws:iam::637423483309:user/farah

Tags:

Environment: Production

Team: Team1

Deployment Script:

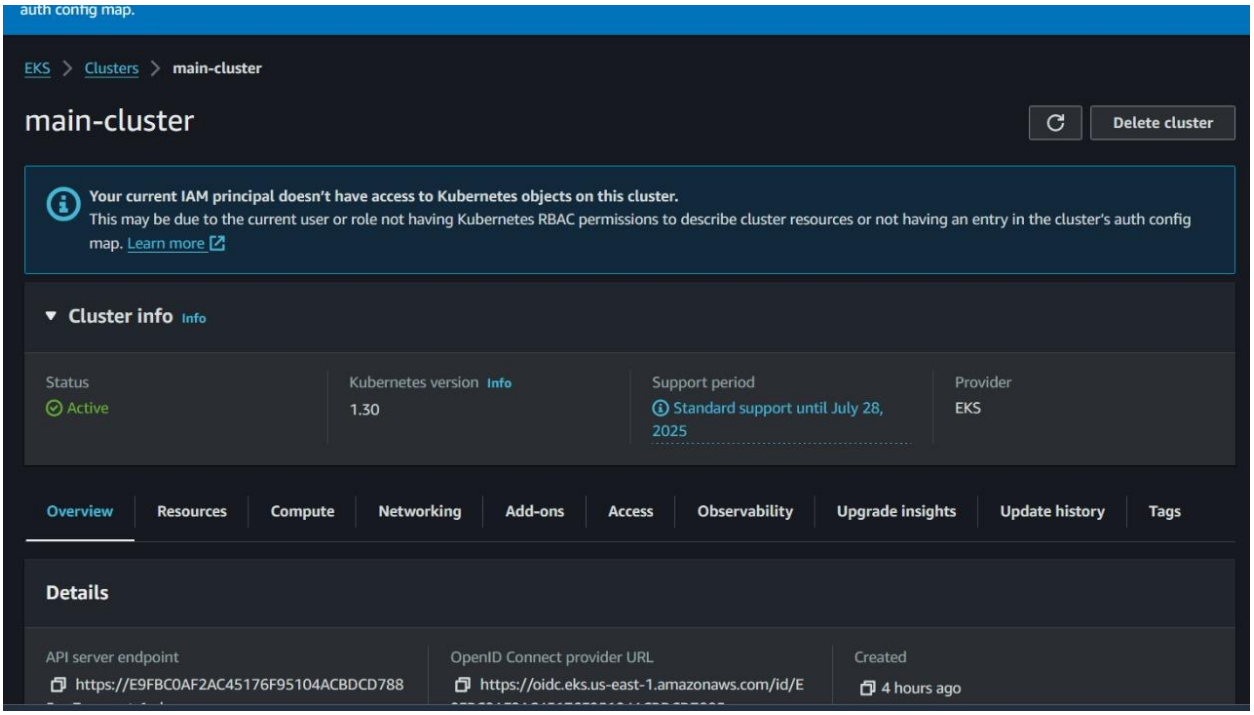
The deploy.sh script automates the deployment process. It builds the Docker images, starts the containers, and cleans up any unused images.

```
#!/bin/bash

# Log in to Docker Hub
docker login

# Retrieve the Docker Hub username
DOCKER_HUB_USERNAME=$(docker info 2>/dev/null | grep -Po '(?<=Username: ).*')

# Check if the username was retrieved successfully
if [ -z "$DOCKER_HUB_USERNAME" ]; then
echo "Failed to retrieve Docker Hub username."
exit 1
fi

echo "Using Docker Hub username: $DOCKER_HUB_USERNAME"

# Define the services
services=("backend" "frontend")

# Remove existing images
for service in "${services[@]}"; do
image_name="$DOCKER_HUB_USERNAME/library-management-system_$service:latest"
```

```
if docker images | grep -q "$image_name"; then
    echo "Removing existing image: $image_name"
    docker rmi -f "$image_name"
fi
done

# Build images using docker-compose
docker-compose build

# Tag and push images for each service
for service in "${services[@]}"; do
# Define image name
image_name="$DOCKER_HUB_USERNAME/library-management-system_$service:latest"

# Tag the image
docker tag "library-management-system_$service:latest" "$image_name"

# Push the image to Docker Hub
docker push "$image_name"
done
```

4.Kubernetes Deployment on EKS:

we deployed an application on Amazon Elastic Kubernetes Service (EKS), started by creating an EKS cluster through the AWS Management Console, configuring the necessary VPC, subnets, and IAM roles. After setting up the cluster, We installed kubectl and aws-iam-authenticator to interact with the Kubernetes API. Then updated the kubeconfig file to connect kubectl to EKS cluster. we defined the application deployment in a YAML file, specifying details like the container image and desired replicas, and applied this configuration using kubectl. To make the application accessible, we created a service YAML file that exposes the application via a LoadBalancer and applied it. Monitored the deployment and services using kubectl commands to ensure they are running correctly. For scaling or updating the deployment, we modified the YAML file and reapplied changes.



5.CI/CD Pipeline Setup:

Setting up a Continuous Integration and Continuous Deployment (CI/CD) pipeline involves automating the process of integrating code changes and deploying applications. We selected a CI/CD tool, such as Jenkins and GitHub Actions. Configured the pipeline to automatically build, test, and deploy code whenever changes are pushed to the repository. This setup typically includes defining stages for build, test, and deployment in the pipeline configuration file. Integrate version control systems to trigger the pipeline upon code commits. Ensuring automated tests are executed to validate code quality before deployment. For deployment, we configured the pipeline to deploy to different environments.

Pipeline Stages:

1.Clean Workspace: Deletes the entire workspace to ensure a clean build environment.

```
    stage('Clean Workspace') {
    steps {
        script {
            deleteDir() // Deletes the entire workspace
        }
    }
}
```

2.Clone Repository: Clones the Library Management System source code from GitHub.

```
    stage('Clone Repository') {
    steps {
        sh 'git config --global http.postBuffer 524288000'
        // Checkout the source code from GitHub
        sh "git clone $GIT_REPO_URL"
    }
}
```

3. Build Docker Images: Builds Docker images for the application using docker-compose.

```
    stage('Build and Push Docker Images') {
    steps {
        script {
            // Build and push Docker images
            sh 'cd Library-Management-System && docker-compose --verbose build'
        }
    }
}
```

4. Tag and Push Images : Tags the Docker images and pushes them to Docker Hub.

```
    stage('Tag and Push Images') {
steps {
    script {
        sh 'docker images'
        // Log in to Docker Hub and push images
        docker.withRegistry("https://index.docker.io/v1/", DOCKER_CREDENTIALS_ID) {
            // Tag and push backend image
            sh """  docker tag library-management-system_backend:latest $DOCKER_REGISTRY/library-management-syst
                    docker push $DOCKER_REGISTRY/library-management-system_backend:latest """

            // Tag and push frontend image
            sh """
                docker tag library-management-system_frontend:latest ${DOCKER_REGISTRY}/library-management-syste
                 docker push ${DOCKER_REGISTRY}/library-management-system_frontend:latest"""

  }
    }
     }
       }
```

5.Terraform Init and Apply: Initializes Terraform and applies the configuration to create and manage infrastructure.

```
    stage('Terraform Init and Apply') {
    steps {
        script {
            // Initialize Terraform

            sh """
            cd cd Library-Management-System/temp
            terraform init

            terraform apply -auto-approve"""
        }
    }
}
```

6. Deploy to Kubernetes: Deploys the Docker images to the EKS cluster.

```
    stage('Deploy to Kubernetes') {
    steps {
        script {
             // Apply Kubernetes configurations
                    sh """
                    cd Library-Management-System/deployment_configurations
                        kubectl apply -f . --validate=false
                        kubectl get services
                    """
        }
    }
}
```