**Devops Training**

1. **Deploy a simple Nginx web server in a Kubernetes cluster.**

Kubectl create nginx-server –image nginx

1. **Create a Kubernetes Deployment for a multi-replica Nginx**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx

replicas: 3 # number of replicas

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

1. **Expose the Nginx Deployment using a Kubernetes Service.**

metadata:

name: nginx-service

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

nodePort: 30000

type: NodePort

1. **Create a Config-Map for storing configuration data and mount it into a Pod.**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx

replicas: 3

template:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

volumeMounts:

- name: conf

mountPath: /etc/nginx/conf.d

volumes:

- name: conf

configMap:

name: nginx-config

items:

- key: nginx.conf

path: default.conf

1. **Create a Docker file to run a WordPress application with a linked MySQL container.**

* Create Docker files for building MYSQL and Wordpress Image

FROM mysql:5.7

ENV MYSQL\_ROOT\_PASSWORD=root

ENV MYSQL\_DATABASE=wordpress

docker build -f Dockerfile-mysql -t my-mysql .

FROM wordpress:latest

docker build -f Dockerfile-wordpress -t my-wordpress .

* Use docker compose for creating the wordpress server and linking it with MYSQL

version: '3'

services:

db:

image: my-mysql

environment:

MYSQL\_ROOT\_PASSWORD: root

MYSQL\_DATABASE: wordpress

wordpress:

image: my-wordpress

ports:

- "8080:80"

depends\_on:

- db

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: root

WORDPRESS\_DB\_PASSWORD: root

docker-compose up -d

1. **Dockerize a Node.js application and expose it on port 3000**

* Dockerfile

# Use an official Node.js runtime as the base image

FROM node:14

# Set the working directory in the container to /app

WORKDIR /app

# Copy package.json and package-lock.json into the container at /app

COPY package\*.json /app/

# Install application dependencies in the container

RUN npm install

# Copy the current directory contents (i.e., your app's source code)

# into the container at /app

COPY . /app/

# Instruct Docker to listen on port 3000 when the container starts

EXPOSE 3000

# Define the command to run the app

CMD [ "node", "server.js" ]

* Creating Image and container

docker build -f Dockerfile-nodejs -t my-nodejs-app .

docker run -it --name my-nodejsapp -p 3000:3000 my-nodejs-app

1. **Create a Dockerfile for a Python web application using Flask. The application should expose port 5000, and when accessed, it should display "Hello, Docker!" in the browser.**

* Dockerfile

FROM python:3.7-slim

# Set the working directory to /app

WORKDIR /app

# Add the current directory contents into the container at /app

ADD . /app

# Install any needed packages specified in requirements.txt

RUN pip install --no-cache-dir -q -r requirements.txt

# Make port 5000 available to the world outside this container

EXPOSE 5000

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

* Dependent files

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

return 'Hello, Docker!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000)

* Requirements

Flask==1.1.2

Werkzeug==1.0.1

Jinja2==2.11.3

itsdangerous==1.1.0

click==7.1.2

MarkupSafe==1.1.1

* Create the docker image and run container

Docker build -f Dockerfile-flask -t my-python-app1

docker run -it --name flaskapp -p 4000:5000 my-python-app1

1. **Create a Terraform script to provision an AWS EC2 instance with an associated security group allowing incoming traffic on port 22 for SSH.**

provider "aws" {

region = "ap-south-1" # Your AWS region

access\_key = "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" # Add your access key

secret\_key = "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" # Add your secret key

}

resource "aws\_security\_group" "allow\_ssh" {

name = "allow\_ssh"

description = "Allow ssh inbound traffic"

ingress {

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

}

resource "aws\_instance" "ec2\_instance" {

ami = "ami-0a7cf821b91bcccbc" # Change this to latest Ubuntu 20.04 LTS AMI ID which can be found on AWS AMI marketplace

instance\_type = "t2.micro" # Choose instance type

key\_name = "devopstraining" # You need to provide your key\_pair name here

vpc\_security\_group\_ids = [aws\_security\_group.allow\_ssh.id]

tags = {

Name = "my\_ec2\_instance"

}

}

1. **Write a Terraform script to provision an AWS RDS instance (MySQL) with specified configurations.**

resource "aws\_db\_instance" "default" {

identifier = "mydbinstance"

allocated\_storage = 20

storage\_type = "gp2"

engine = "mysql"

engine\_version = "5.7"

instance\_class = "db.t2.micro"

name = "mydb"

username = "foo"

password = "foobarbaz"

parameter\_group\_name = "default.mysql5.7"

skip\_final\_snapshot = true

publicly\_accessible = true

vpc\_security\_group\_ids = ["sg-0414175f86957415b"]

}