**Final\_Assement\_Terraform\_Task**

**BY - Sheshadri Reddy**

**1. Using aws terraform code to deploy sample nginx/python applications. -**

* **Build docker image for sample nginx/python and push to aws elastic container registry(ECR) using docker cli.**

**- Create VPC with 1 public and private subnet, Security Groups and EC2 instance using terraform.**

**- Deploy Nginx Application as Docker Container using User Data script using terraform. Image should be pulled from ECR.**

**- Outcome is able to access the nginx website with the url http://<public\_ip>:<port>.**

**- Keep the code stuff into the git repository.**

**- Document all the execution steps.**

**1. Build docker image for sample nginx/python and push to aws elastic container registry(ECR) using docker cli.**

**FROM nginx**

**WORKDIR /opt**

**RUN apt-get update && apt-get upgrade -y**

**COPY index.html /usr/share/nginx/html**

**CMD ["nginx", "-g", "daemon off"]**

**# FROM nginx:latest**

**EXPOSE 8080**

**CMD ["nginx", "-g", "daemon off;"]**

**DOCKER FILE**

**--------------------**

* **It's a file using which we can create a docker image. In the docker file we are going to write instructions to create an image quickly.**
* **In the docker file we are going to use docker DSL(Domain specific language).**

**Keywords Used**

**1.FROM**

**2.WORKDIR**

**3.RUN**

**4.COPY**

**5.CMD**

* **By using all the above Keywords we can build the Docker Image**

**Docker Image**

**It's an executable package which contains everything (app code and its dependencies) such as softwares, configurations, env, vars etc.) which is required to run a piece of code.**

**1.FROM**

**FROM Nginx**

* **It's a base image using which we are creating our image.**
* **Pulled Nginx Image from the DockerHub**

**2.WORKDIR**

**WORKDIR /OPT**

* **The WORKDIR Instruction in a Dockerfile is used to set working dir for any subsequent instructions during the image build process.It allows you to specify a directory where commands such as “RUN” , “COPY”, “ADD” Will be Executed.**
* **Set the WORKINGDIR for this project/Image.**

**3.RUN**

* **using run we can run any command on top of the previous layer or image.**
* **These RUN commands will be executed while creating an image.**
* **We can have a number of RUN instructions.will process all the instructions from top to bottom.**
* **Here i Have installed a Nginx using the Run**
* **RUN apt-get update && apt-get upgrade -y**

**4. COPY**

* **Using copy we can copy local files/folders from the system where we're building the image,(Local system) to the image.**
* **We can copy Regular Files only.**
* **The File should be part of our build context only.**
* **COPY index.html /usr/share/nginx/html**
* **I have copied the “index.html” from the directory where the dockerfile exists. To /usr/share/nginx/html directory inside the docker image.**

**5.CMD**

* **Using CMD also we can run the Commands.**
* **CMD instruction or command will execute the command while creating the container.**
* **This is used to run an application( process).**

**CMD ["nginx", "-g", "daemon off"]**

* **This is the default command to run when the container starts**
* **Nginx : This command is used to start the server**
* **-g : It allows nginx configuration directly on to the Command line**

**- Demon off: it allows you to see its log output and interact with it directly in the same terminal session. This is useful for debugging or monitoring purposes, as you can observe the log messages and any potential errors or warnings in real-time.**

**Build the Image**

**Docker build -t nginx-container .**

**Too list the images**

**Docker images**

**Run the container with image**

**Docker run -d -it <img-id>/<img-name>**

**push to aws elastic container registry(ECR) using Docker CLI**

**Elastic Container Registry**

**Elastic Container Registry (ECR) is a fully managed container registry service provided by Amazon Web Services (AWS). It is used to store, manage, and deploy container images.**

**To create a Repo**

* **Login to aws —-> Aws Console —>search ECR —-> Create Repo → Repo\_Name**
* **Aws Configure → Configure your Acces\_key → secret\_key**
* **Login in to the aws using this command**

**aws ecr get-login-password --region ap-south-1 | docker login --username AWS --password-stdin 999841539932.dkr.ecr.ap-south-1.amazonaws.com**

* **After the build is completed, tag your image so you can push the image to this repository:**

**Pusdocker tag nginx-conatiner:latest 999841539932.dkr.ecr.ap-south-1.amazonaws.com/new:latesth the image to ecr repository**

* **Run the following command to push this image to your newly created AWS repository**

**docker push 999841539932.dkr.ecr.ap-south-1.amazonaws.com/new:latest**

**2. Create VPC with 1 public and private subnet, Security Groups and EC2 instance using terraform.**

**# Configure AWS provider**

**provider "aws" {**

**region = "ap-south-1"**

**}**

**Provider AWS : This specifies that the provider being configured is for AWS.**

**Region : This line specifies the AWS region where resources will be created.**

**# Create VPC**

**resource "aws\_vpc" "my\_vpc" {**

**cidr\_block = "10.0.0.0/16"**

**tags = {**

**Name = "my-vpc"**

**}**

**}**

**resource "aws\_vpc" "my\_vpc" : This line specifies the resource block type as "aws\_vpc" and assigns a name "my\_vpc".**

**Cidr\_block: The CIDR block determines the IP address range that the VPC will use. In this example, the VPC will have the IP address range of 10.0.0.0/16, which allows for a maximum of 65,536 IP addresses.**

**Tags : This line sets a tag for the VPC resource. Tags are key-value pairs used to label and organize resources in AWS.**

**# Create public subnet**

**resource "aws\_subnet" "public\_subnet" {**

**vpc\_id = aws\_vpc.my\_vpc.id**

**cidr\_block = "10.0.1.0/24"**

**availability\_zone = "ap-south-1a"**

**map\_public\_ip\_on\_launch = true**

**tags = {**

**Name = "public-subnet"**

**}**

**}**

**resource "aws\_subnet" "public\_subnet": This line specifies the resource block type as "aws\_subnet" and assigns a name "public\_subnet" to this resource block. The name "public\_subnet" is used to reference this subnet resource elsewhere in the Terraform configuration.**

**vpc\_id = aws\_vpc.my\_vpc.id: This line specifies the VPC ID to which the subnet will be associated. It uses the id attribute of the previously created VPC resource (aws\_vpc.my\_vpc.id) to establish the association.**

**cidr\_block = "10.0.1.0/24": This line defines the CIDR block for the subnet. The CIDR block determines the IP address range that the subnet will use. In this example, the subnet will have the IP address range of 10.0.1.0/24, allowing for a maximum of 256 IP addresses.**

**availability\_zone = "ap-south-1a": This line specifies the availability zone in which the subnet will be created. Availability zones are distinct locations within an AWS region that provide high availability and fault tolerance. In this case, the subnet will be created in the availability zone "ap-south-1a" of the AWS region.**

**map\_public\_ip\_on\_launch = true: This line indicates that instances launched in this subnet should be assigned a public IP address. By setting this to true, instances launched in this subnet will have the ability to communicate directly with the internet.**

**tags = { Name = "public-subnet" }: This line sets a tag for the subnet resource. Similar to the VPC tag, it assigns a tag with the key "Name" and the value "public-subnet" to the subnet resource for better organization and identification.**

**# Create private subnet**

**resource "aws\_subnet" "private\_subnet" {**

**vpc\_id = aws\_vpc.my\_vpc.id**

**cidr\_block = "10.0.2.0/24"**

**availability\_zone = "ap-south-1a"**

**tags = {**

**Name = "private-subnet"**

**}**

**}**

**resource "aws\_subnet" "private\_subnet": This line specifies the resource block type as "aws\_subnet" and assigns the name "private\_subnet" to this resource block. The name "private\_subnet" is used to reference this subnet resource elsewhere in the Terraform configuration.**

**vpc\_id = aws\_vpc.my\_vpc.id: This line specifies the VPC ID to which the subnet will be associated. It uses the id attribute of the previously created VPC resource (aws\_vpc.my\_vpc.id) to establish the association.**

**cidr\_block = "10.0.2.0/24": This line defines the CIDR block for the subnet. The CIDR block determines the IP address range that the subnet will use. In this example, the subnet will have the IP address range of 10.0.2.0/24, allowing for a maximum of 256 IP addresses.**

**availability\_zone = "ap-south-1a": This line specifies the availability zone in which the subnet will be created. It indicates that the subnet will be located in the availability zone "ap-south-1a" of the AWS region.**

**tags = { Name = "private-subnet" }: This line sets a tag for the subnet resource. It assigns a tag with the key "Name" and the value "private-subnet" to the subnet resource, helping with better organization and identification.**

**# Create security group for EC2 instance**

**resource "aws\_security\_group" "instance\_sg" {**

**name = "instance-sg"**

**description = "Security group for EC2 instance"**

**vpc\_id = aws\_vpc.my\_vpc.id**

**ingress {**

**from\_port = 22**

**to\_port = 22**

**protocol = "tcp"**

**cidr\_blocks = ["0.0.0.0/0"] # Replace with your desired source IP range**

**}**

**egress {**

**from\_port = 0**

**to\_port = 0**

**protocol = "-1"**

**cidr\_blocks = ["0.0.0.0/0"]**

**}**

**tags = {**

**Name = "instance-sg"**

**}**

**}**

**resource "aws\_security\_group" "instance\_sg": This line specifies the resource block type as "aws\_security\_group" and assigns the name "instance\_sg" to this resource block. The name "instance\_sg" is used to reference this security group resource elsewhere in the Terraform configuration.**

**name = "instance-sg": This line defines the name of the security group as "instance-sg". You can provide a custom name for your security group.**

**description = "Security group for EC2 instance": This line provides a description for the security group, which helps in identifying its purpose or function.**

**vpc\_id = aws\_vpc.my\_vpc.id: This line specifies the VPC ID to which the security group will be associated. It uses the id attribute of the previously created VPC resource (aws\_vpc.my\_vpc.id) to establish the association.**

**ingress { ... }: This block specifies the inbound rules for the security group. In this example, there is one ingress rule allowing TCP traffic on port 22 (SSH) from any source IP address (0.0.0.0/0). You can modify the from\_port, to\_port.**

**egress { ... }: This block specifies the outbound rules for the security group. In this example, there is one egress rule allowing all traffic (protocol = "-1") to any destination IP address (0.0.0.0/0). You can modify the from\_port, to\_port, protocol, and cidr\_blocks values as needed.**

**tags = { Name = "instance-sg" }: This line sets a tag for the security group resource. It assigns a tag with the key "Name" and the value "instance-sg" to the security group for better organization and identification.**

**# Create EC2 instance**

**resource "aws\_instance" "my\_instance" {**

**ami = "ami-057752b3f1d6c4d6c"**

**instance\_type = "t2.micro"**

**subnet\_id = aws\_subnet.public\_subnet.id**

**vpc\_security\_group\_ids = [aws\_security\_group.instance\_sg.id]**

**user\_data = <<-EOF**

**#!/bin/bash**

**echo "Hello, World!" > index.html**

**nohup python -m SimpleHTTPServer 80 &**

**EOF**

**tags = {**

**Name = "my-instance"**

**}**

**}**

**resource "aws\_instance" "my\_instance": This line specifies the resource block type as "aws\_instance" and assigns the name "my\_instance" to this resource block.**

**ami = "ami-057752b3f1d6c4d6c": This line specifies the AMI (Amazon Machine Image) ID for the EC2 instance.**

**instance\_type = "t2.micro": This line defines the instance type, which determines the hardware specifications of the EC2 instance. In this example, "t2.micro" is used, which represents a small, low-cost instance type suitable for testing and development.**

**subnet\_id = aws\_subnet.public\_subnet.id: This line specifies the subnet ID where the EC2 instance will be launched. It uses the id attribute of the previously created public subnet resource (aws\_subnet.public\_subnet.id) to identify the subnet.**

**vpc\_security\_group\_ids = [aws\_security\_group.instance\_sg.id]: This line assigns the security group ID(s) to the EC2 instance. It uses the id attribute of the previously created security group resource (aws\_security\_group.instance\_sg.id) to associate the security group with the instance.**

**user\_data = <<-EOF ... EOF: This block contains the user data script that will be executed on the EC2 instance when it starts. User data allows you to run scripts and perform configurations on the instance at launch time. In this example, a bash script is provided that writes "Hello, World!" to an index.html file and starts a SimpleHTTPServer on port 80.**

**tags = { Name = "my-instance" }: This line sets a tag for the EC2 instance resource. It assigns a tag with the key "Name" and the value "my-instance".**

**Terraform Init :: Used to Initialize the terraform working Directory**

**Terraform Plan:: It is used to create an Execution Plan applying changes to your infrastructure based on our configuration.**

**Terraform Apply:: Terraform apply is used to apply the changes defined in your terraform configuration.**

**3.Deploy Nginx Application as Docker Container using User Data script using terraform. Image should be pulled from ECR.**

**provider "aws" {**

**region = "ap-south-1"**

**}**

**resource "aws\_vpc" "myVpc1" {**

**cidr\_block = "10.0.0.0/24"**

**}**

**data "aws\_availability\_zones" "available\_zones" {}**

**resource "aws\_subnet" "publicSubnet1" {**

**vpc\_id = aws\_vpc.myVpc1.id**

**cidr\_block = "10.0.0.0/25"**

**availability\_zone = data.aws\_availability\_zones.available\_zones.names[0]**

**tags = {**

**Name = "publicSubnet1"**

**}**

**}**

**resource "aws\_subnet" "privateSubnet1" {**

**vpc\_id = aws\_vpc.myVpc1.id**

**cidr\_block = "10.0.0.128/25"**

**availability\_zone = data.aws\_availability\_zones.available\_zones.names[1]**

**tags = {**

**Name = "privateSubnet1"**

**}**

**}**

**resource "aws\_internet\_gateway" "myIGW1" {**

**vpc\_id = aws\_vpc.myVpc1.id**

**tags = {**

**Name = "myIGW1"**

**}**

**}**

**resource "aws\_route\_table" "myPublicRoute" {**

**vpc\_id = aws\_vpc.myVpc1.id**

**route {**

**cidr\_block = "0.0.0.0/0"**

**gateway\_id = aws\_internet\_gateway.myIGW1.id**

**}**

**tags = {**

**Name = "myRoute"**

**}**

**}**

**// associate subnet with route table**

**resource "aws\_route\_table\_association" "myPublicRouteAssociate" {**

**subnet\_id = aws\_subnet.publicSubnet1.id**

**route\_table\_id = aws\_route\_table.myPublicRoute.id**

**}**

**resource "aws\_security\_group" "mySecureGrp" {**

**name = "mySecureGrp"**

**vpc\_id = aws\_vpc.myVpc1.id**

**ingress {**

**from\_port = 22**

**to\_port = 22**

**protocol = "tcp"**

**cidr\_blocks = ["0.0.0.0/0"]**

**}**

**ingress {**

**from\_port = 8080**

**to\_port = 8080**

**protocol = "tcp"**

**cidr\_blocks = ["0.0.0.0/0"]**

**}**

**ingress {**

**from\_port = 443**

**to\_port = 443**

**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"]**

**//ipv6\_cidr\_blocks = ["::/0"]**

**}**

**tags = {**

**Name = "mySecureGrp"**

**}**

**}**

**resource "aws\_instance" "myEc2Public" {**

**ami = "ami-0f5ee92e2d63afc18"**

**instance\_type = "t2.micro"**

**key\_name = "Terraform"**

**subnet\_id = aws\_subnet.publicSubnet1.id**

**vpc\_security\_group\_ids = [aws\_security\_group.mySecureGrp.id]**

**associate\_public\_ip\_address = true**

**user\_data = <<-EOF**

**#! /bin/bash**

**echo "hello world!" > hello.txt**

**sudo apt-get update -y**

**sudo apt install docker.io -y**

**curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"**

**unzip awscliv2.zip**

**sudo ./aws/install**

**aws ecr-public get-login-password --region us-east-1 | sudo docker login --username AWS --password-stdin public.ecr.aws/b9c2h9h8**

**sudo docker pull public.ecr.aws/b9c2h9h8/gayu\_repo1:latest**

**sudo docker run -d -p 8080:80 public.ecr.aws/b9c2h9h8/gayu\_repo1:latest**

**EOF**

**tags = {**

**Name = "fi-image”**

**}**

**echo "hello world!" > hello.txt: This line writes the text "hello world!" to a file named "hello.txt". It demonstrates a simple command to create a file.**

**sudo apt-get update -y: This line updates the package lists for installed packages on the EC2 instance. The -y flag automatically answers "yes" to any prompts during the update process.**

**sudo apt install docker.io -y: This line installs Docker on the EC2 instance. The package manager apt is used to download and install Docker. The -y flag automatically answers "yes" to any prompts during the installation process.**

**curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip": This line uses curl to download the AWS Command Line Interface (CLI) installation package.**

**unzip awscliv2.zip: This line unzips the downloaded AWS CLI package.**

**sudo ./aws/install: This line runs the installation script for the AWS CLI, installing it on the EC2 instance.**

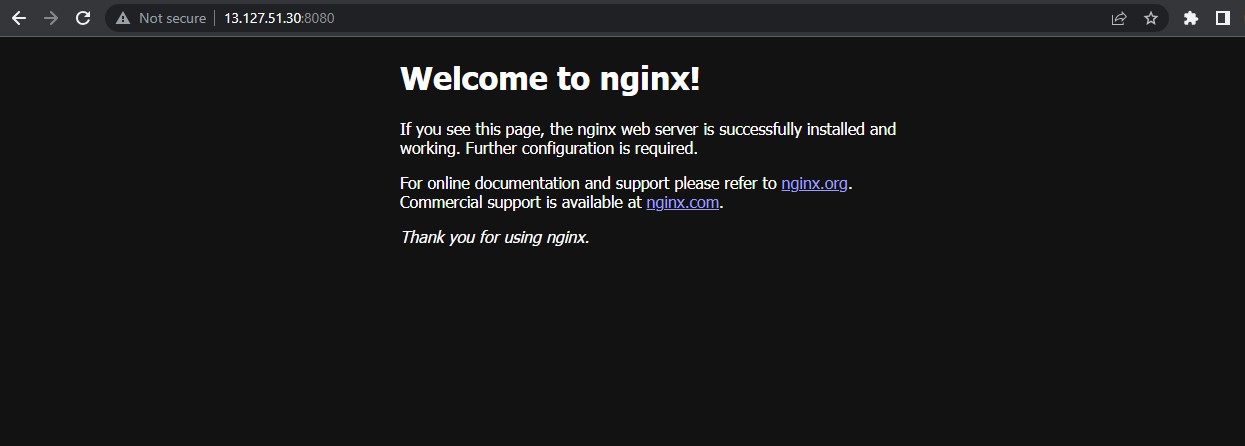
**aws ecr-public get-login-password --region us-east-1 | sudo docker login --username AWS --password-stdin public.ecr.aws/b9c2h9h8: This line retrieves an authentication token from AWS ECR Public to authenticate the Docker client with the ECR repository. It uses the AWS CLI command get-login-password to obtain the password and then logs in to the ECR repository using the docker login command.**

**sudo docker pull public.ecr.aws/b9c2h9h8/gayu\_repo1:latest: This line pulls the Docker image named "gayu\_repo1" from the ECR repository located at public.ecr.aws/b9c2h9h8. The :latest tag indicates the latest version of the image.**

**sudo docker run -d -p 8080:80 public.ecr.aws/b9c2h9h8/gayu\_repo1:latest: This line runs a Docker container in detached mode (-d) based on the pulled image. The container is bound to port 8080 on the EC2 instance and maps it to port 80 within the container. This allows access to the web service provided by the container.**

**Outcome is able to access the nginx website with url http://<public\_ip>:<port>.**

**http://13.127.51.30:8080/**

****

**Keep the code stuff into git repository.**

**REPO URL: https://github.com/sheshadriReddy90/Final\_assement\_task**