**DevOps Training Tasks**

Notes:

• Create a new repository in your GitHub account.

• Push all files, scripts, images, etc to this newly created repository for

each task, folder structure in repository should look like below:

o Repo-devops-tasks

▪ Task-1

• ReadME.md

• File1

• File2

▪ Task-2

• ReadME.md

• File1

• File2

• You need to create a simple document for each task and add steps

followed for each task and you should add the screenshot of output of

the result.

• All the documents need to be created in Markdown format. Learn how

to create document in markdown format.

• Important Note: Do not push your AWS Access key and Secret key to

GitHub repository, also do not push sensitive data like credentials, keys,

etc.

**Task 1: Docker, Docker Hub**

• Create a Dockerfile for a simple web application (e.g. a Node.js or

Python app)

• Build the image using the Dockerfile and run the container

• Verify that the application is working as expected by accessing it in a

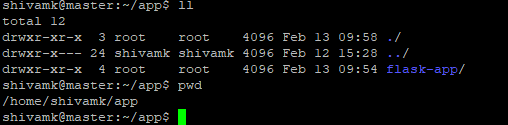
web browser

• Push the image to a public or private repository (e.g. Docker Hub)

https://tecadmin.net/how-to-create-and-run-a-flask-application-using-docker/#:~:text=Create%20a%20Basic%20Flask%20Application,locally%20or%20on%20a%20server.&text=Next%2C%20create%20the%20Python%20virtual%20environment%20and%20then%20activate%20the%20environment.&text=Now%20install%20the%20Flask%20python%20module%20under%20the%20virtual%20environment.

sudo mkdir flask-app

cd flask-app/



sudo apt install python3.10-venv

python3 -m venv venv1

sudo python3 -m venv venv1

source venv1/bin/activate

apt install python3-pip

pip install Flask

pip freeze > requirements.txt

cd flask-app/

vim app.py

from flask import Flask

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

@app.route('/')

def index():

return 'Hello'

@app.route('/welcome')

def welcome():

return 'Welcome'

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

app.run(debug=True)

sudo vi Dockerfile

FROM python:3-alpine

# Create app directory

WORKDIR /app

# Install app dependencies

COPY requirements.txt ./

RUN pip install -r requirements.txt

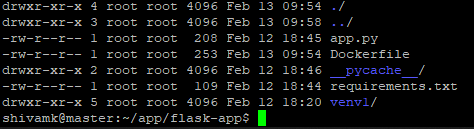
# Bundle app source

COPY . .

EXPOSE 5000

CMD [ "flask", "run","--host","0.0.0.0","--port","5000"]

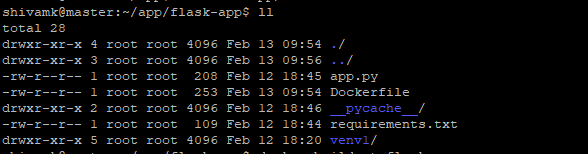
cd flask-app/

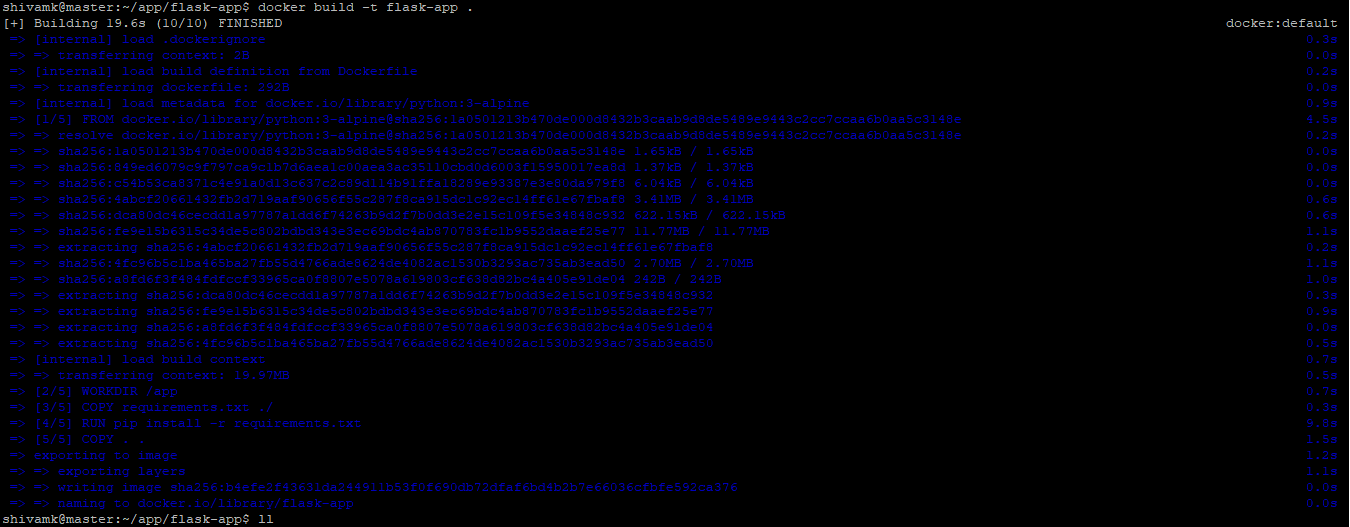


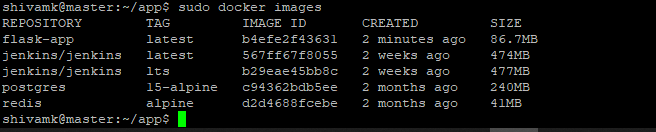
flask run --host 0.0.0.0 --port 5000

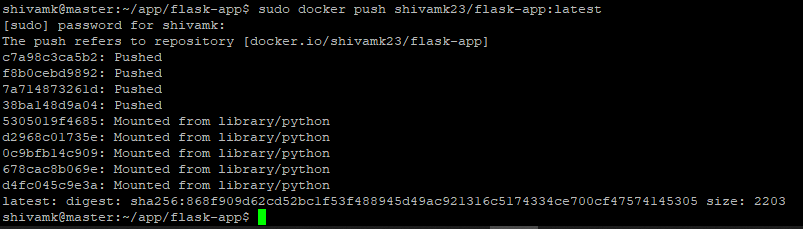
docker build -t flask-app .

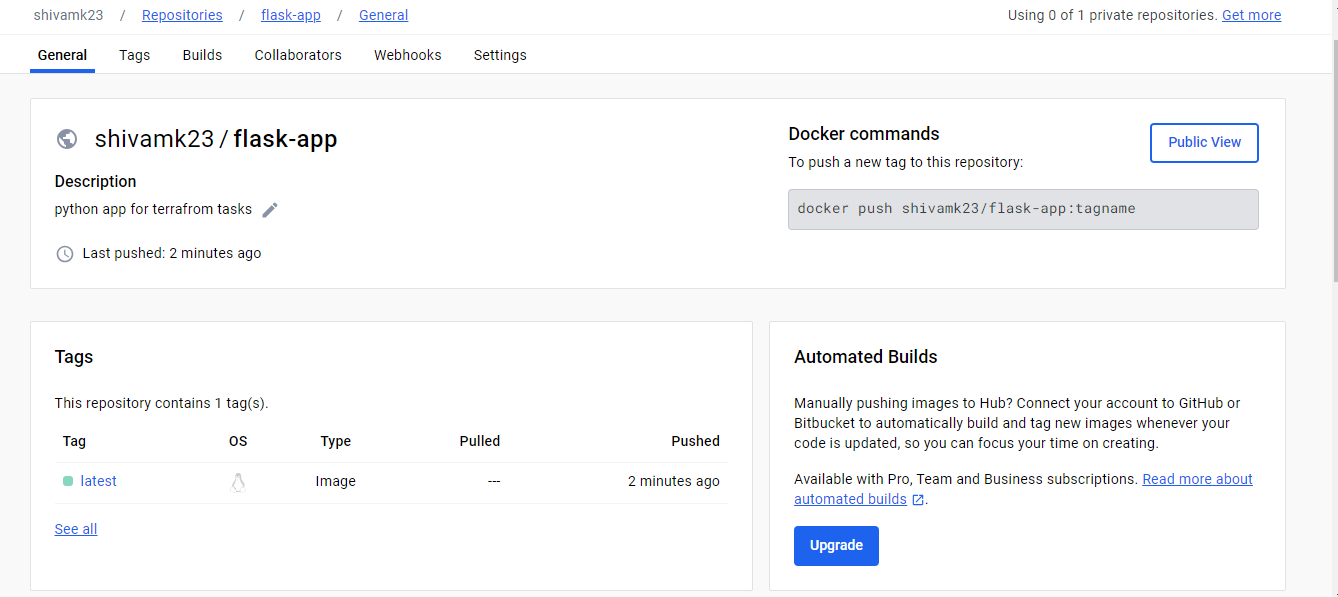
sudo docker images











**Task 2: AWS, Jenkins, Docker, CICD, NodeJS**

• Set up Jenkins on AWS EC2 instance.

• Create a security group for Jenkins.

• Connect to Jenkins instance using EC2 security groups

• Install Jenkins on the EC2 instance

## How to run Jenkins as Docker in Docker

To allow a Jenkins Docker container to run commands outside the container on the host operating system, you typically need to mount the Docker socket from the host into the Jenkins container. This allows the Jenkins container to communicate with the Docker daemon running on the host.

Here's how you can achieve this:

1. **Run Jenkins Container with Docker Socket Mount**:

When running the Jenkins container, mount the Docker socket as a volume inside the container. This can be done by adding the **-v** option to the **docker run** command.

bashCopy code

docker run -d -v /var/run/docker.sock:/var/run/docker.sock jenkins/jenkins:<tag>

Replace **<tag>** with the version of the Jenkins image you want to use.

1. **Adjust Jenkins User Permissions**:

By default, Jenkins runs as the user **jenkins** inside the container. You may need to ensure that this user has the necessary permissions to access the Docker socket.

bashCopy code

sudo usermod -aG docker jenkins

This command adds the **jenkins** user to the **docker** group, allowing it to execute Docker commands.

1. **Install Docker in Jenkins Container (if needed)**:

If the Jenkins container doesn't have Docker installed, you may need to install it manually inside the container. You can do this by accessing the Jenkins container and installing Docker using the appropriate package manager.

bashCopy code

docker exec -it <container\_id> /bin/bash

Once inside the container, install Docker. The exact commands depend on the Linux distribution used in the Jenkins container.

1. **Configure Jenkins**:

Inside Jenkins, you'll need to configure it to use Docker as the build agent. This can be done by installing the Docker Pipeline plugin and configuring Jenkins pipelines to use Docker agents.

1. **Test the Configuration**:

After setting up Jenkins and configuring the Docker integration, you can create a test pipeline job that runs Docker commands (e.g., **docker ps**) to verify that Jenkins can communicate with the Docker daemon running on the host.

By following these steps, you'll enable the Jenkins container to run commands outside the container on the host operating system via the Docker socket mount. Remember to consider security implications when granting Jenkins container access to the Docker socket, as it effectively grants it root-level access to the host system.

User

sudo docker run -d -p 8090:8080 -p 50000:50000 -v jenkins\_home:/var/jenkins\_home -v /var/run/docker.sock:/var/run/docker.sock --name jenkins\_new jenkins/Jenkins

• Creating a Dockerfile (you can use Dockerfile created in Task1)

• Install and manage some Jenkins plugins

• Create a job for automating CI/CD deployment

• Integrate Jenkins and GitHub

• Refer this blog for this task:

https://devopscommunity.hashnode.dev/deploy-a-nodejs-app-using-jenkins-on-aws-ec2-instances

<https://devopscommunity.hashnode.dev/deploy-a-nodejs-app-using-jenkins-on-aws-ec2-instances>

https://www.jenkins.io/doc/tutorials/tutorial-for-installing-jenkins-on-AWS/

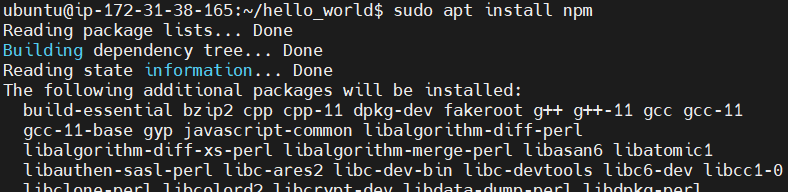
NODE JS Hello World::

1. **Install Node.js for your platform (MacOS, Windows or Linux)**

**nodejs.org**

sudo apt install nodejs

sudo apt install npm

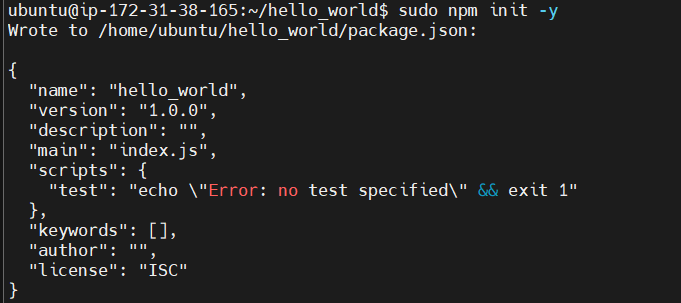


Creating a simple "Hello, World!" Node.js web application involves several steps:

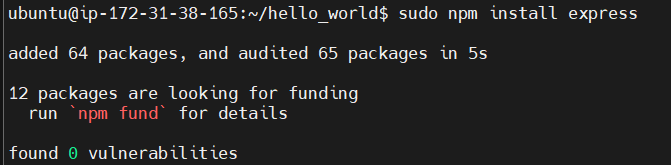
1. **Initialize the project**:
   * Create a new directory for your project.



* + Open a terminal or command prompt and navigate to the directory you just created.
  + Run **npm init -y** to initialize a new Node.js project with default settings.

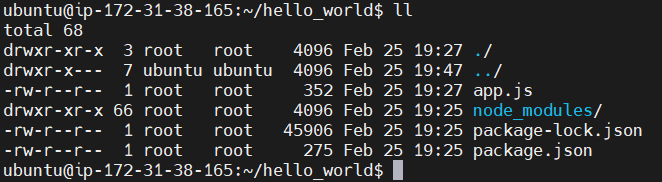


1. **Install Express** (optional, but recommended for web applications):
   * Express is a popular web framework for Node.js that simplifies the process of creating web applications.
   * Run **npm install express** to install Express as a dependency for your project.



1. **Create the main application file**:
   * Create a JavaScript file (e.g., **app.js**) in your project directory.
   * Open **app.js** in a text editor.
   * Write the code for your "Hello, World!" web application using Express. Below is a simple example:





// Import the Express module

const express = require('express');

// Create an Express application

const app = express();

// Define a route handler for the root URL

app.get('/', (req, res) => {

res.send('Hello, World!');

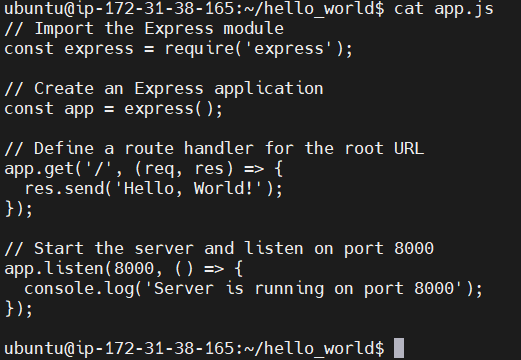
});

// Start the server and listen on port 8000

app.listen(8000, () => {

console.log('Server is running on port 8000');

});

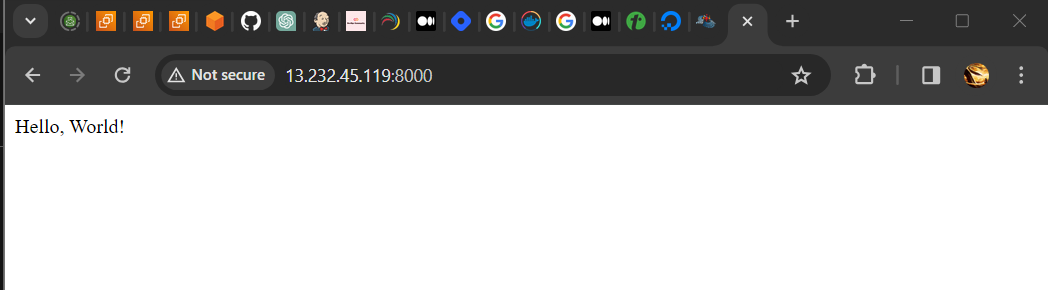


1. **Run the application**:
   * In the terminal, navigate to your project directory.
   * Run **node app.js** to start the Node.js application.

node app.js



1. **Access the application**:
   * Open a web browser and navigate to **http://localhost:3000**.
   * You should see the message "Hello, World!" displayed in the browser.



To create a Dockerfile for your "Hello, World!" Node.js web application, you'll follow these steps:

1. **Create a Dockerfile**: In your project directory, create a file named **Dockerfile**.
2. **Open the Dockerfile in a text editor** and add the following content:

sudo vi Dockerfile



DockerfileCopy code

# Use the official Node.js image with the LTS version

FROM node:14

# Set the working directory inside the container

WORKDIR /app

# Copy package.json and package-lock.json to the working directory

COPY package\*.json ./

# Install dependencies

RUN npm install

# Copy the rest of the application code to the working directory

COPY . .

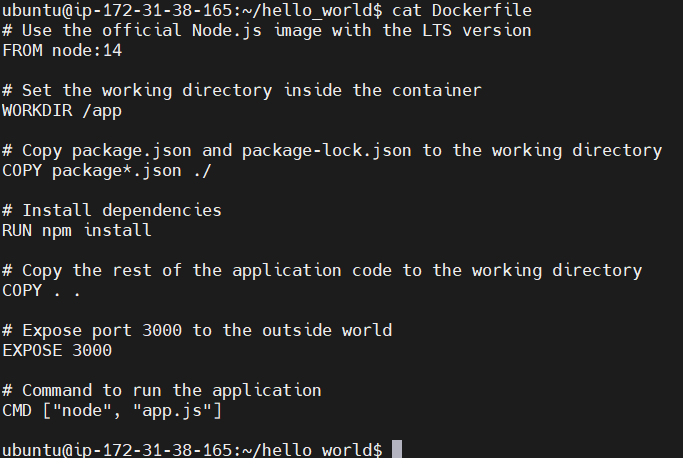
# Expose port 3000 to the outside world

EXPOSE 3000

# Command to run the application

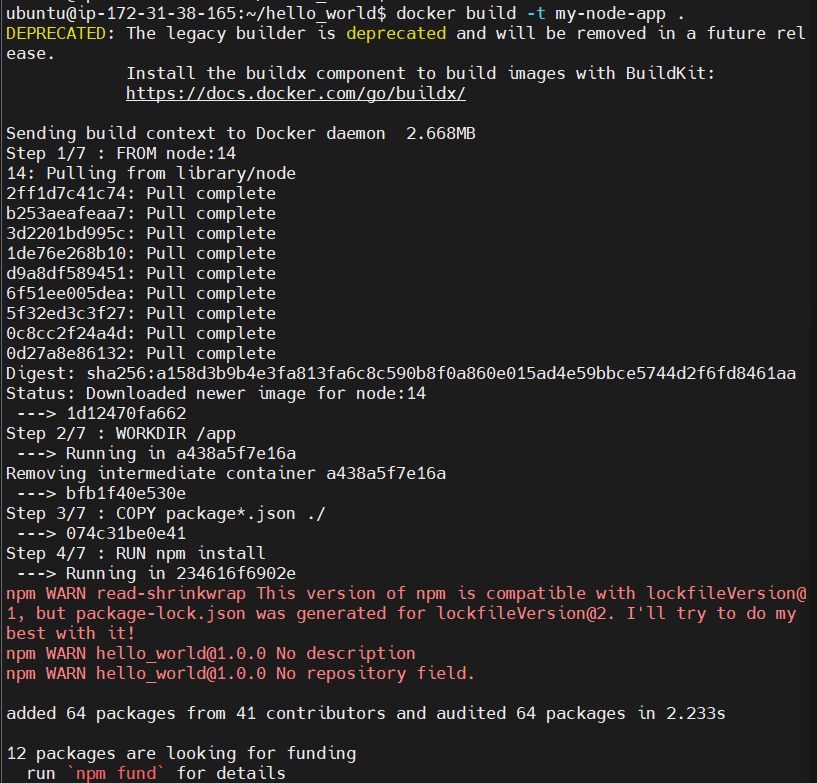
CMD ["node", "app.js"]

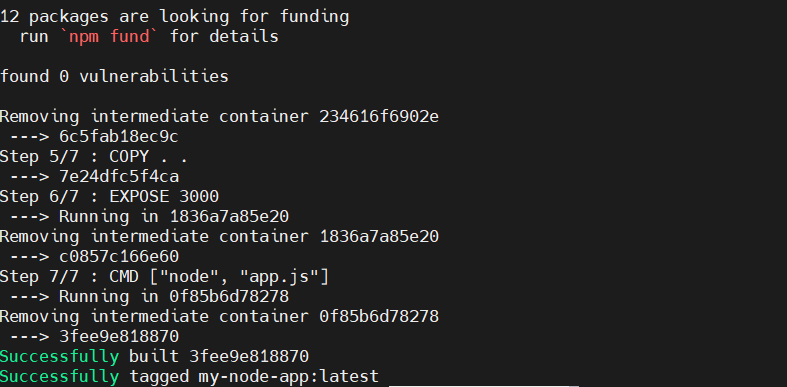
1. **Save the Dockerfile**.



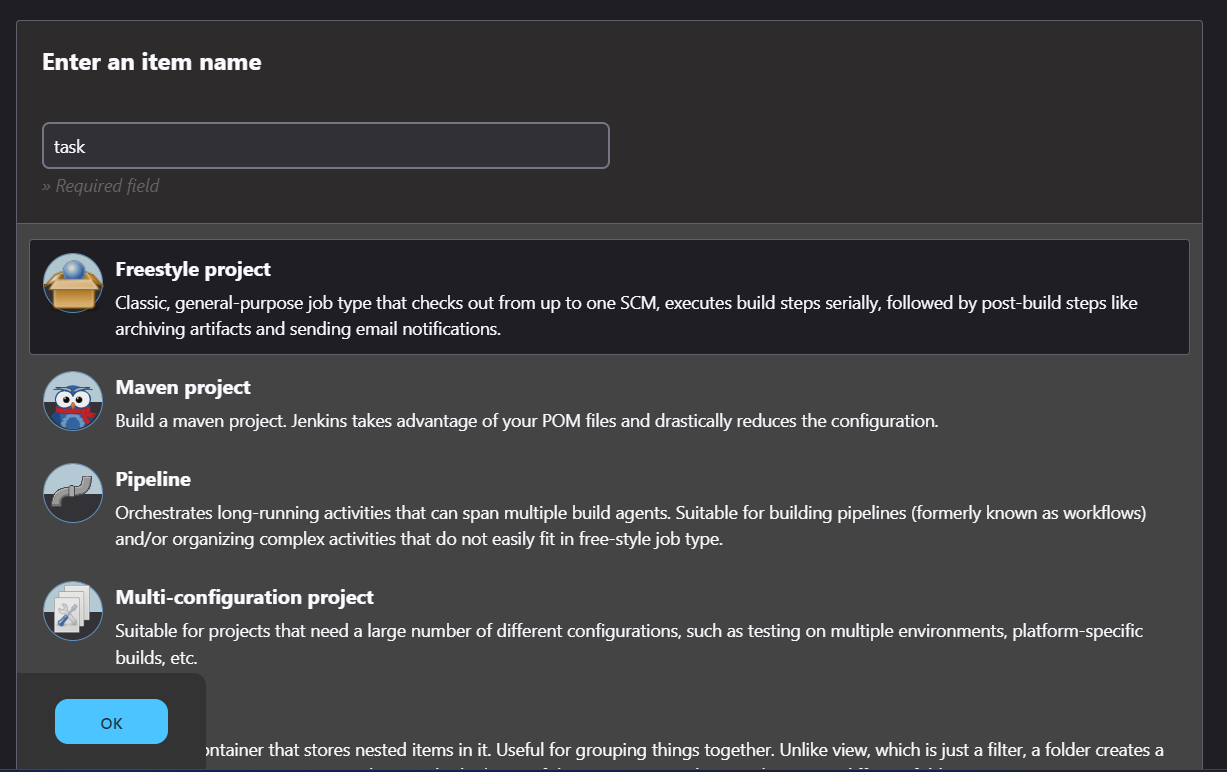
This Dockerfile specifies a multi-stage build process:

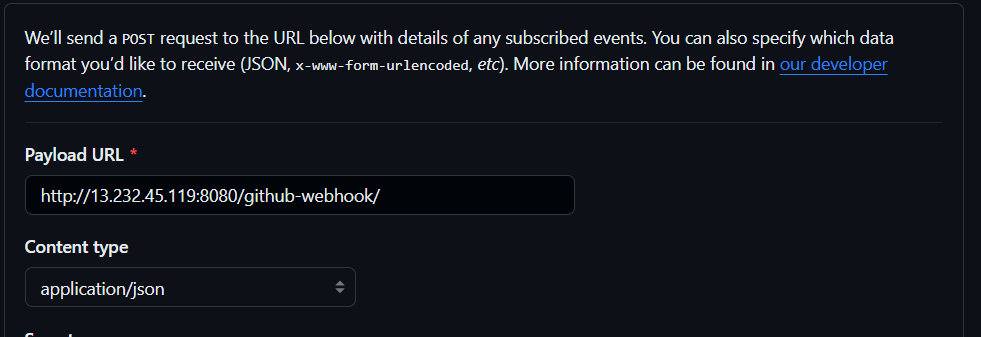
* It uses the official Node.js image with the LTS version as the base image.
* It sets the working directory inside the container to **/app**.
* It copies **package.json** and **package-lock.json** to the working directory and installs dependencies using **npm install**.
* It copies the rest of the application code to the working directory.
* It exposes port **3000** of the container to the outside world.
* It specifies the command to run the application (**node app.js**).

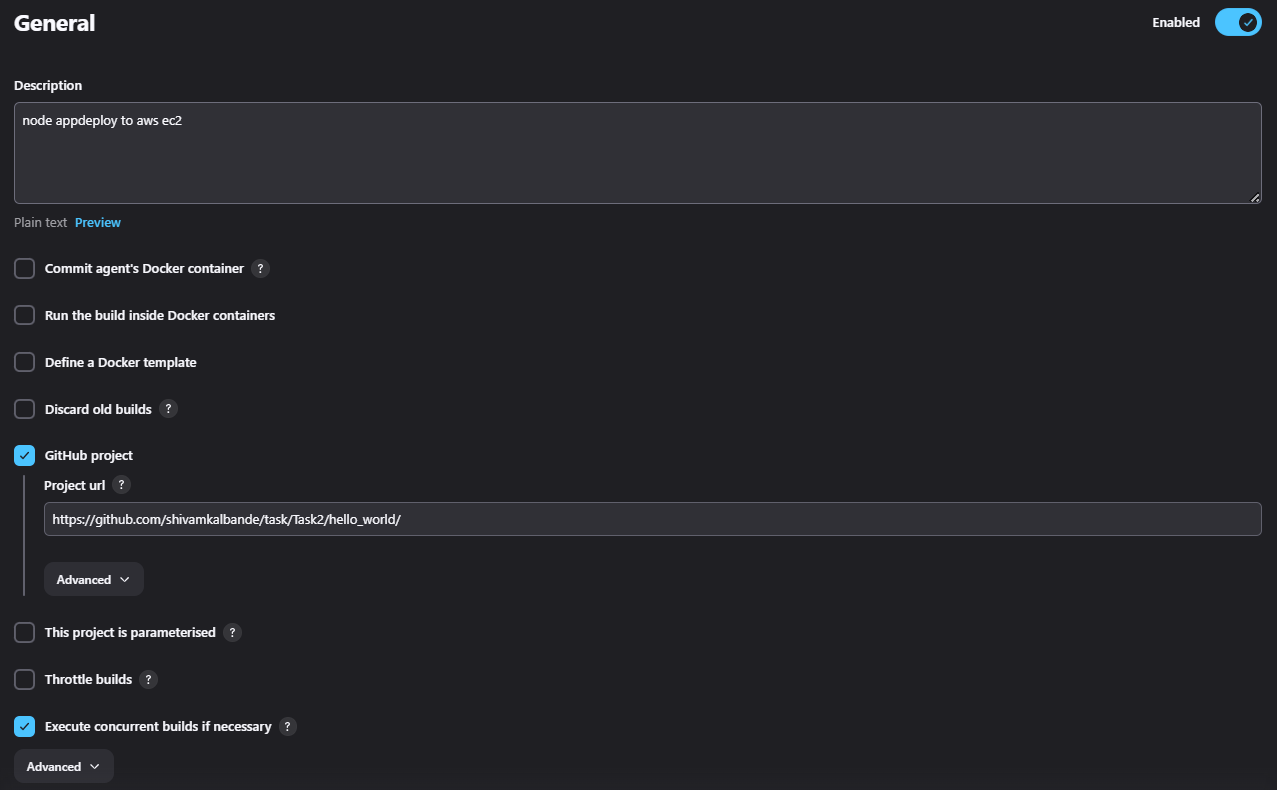


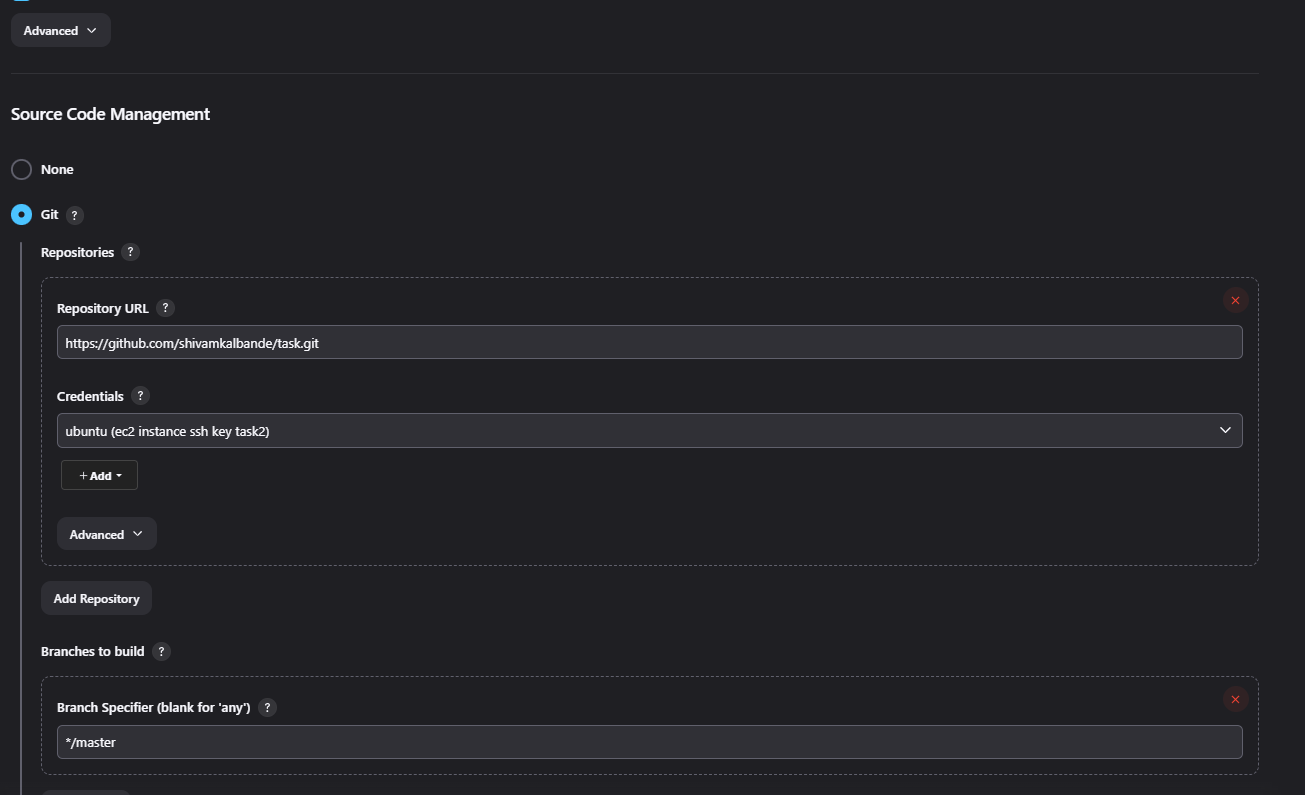


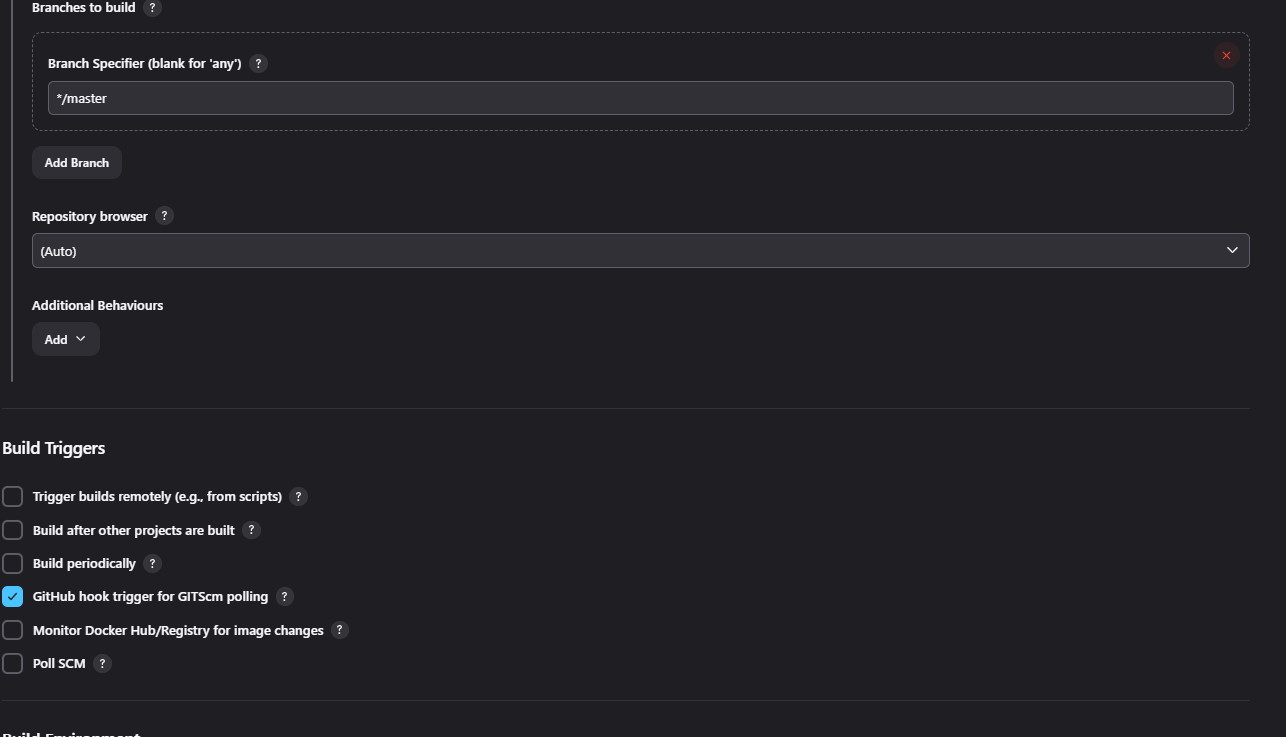
With this Dockerfile, you can now build a Docker image for your Node.js application by running **docker build .** in the project directory. After the image is built, you can run a container based on the image using **docker run -p 3000:3000 <image\_id>**. This will start your Node.js application inside a Docker container, and you'll be able to access it at [**http://localhost:3000**](http://localhost:3000).

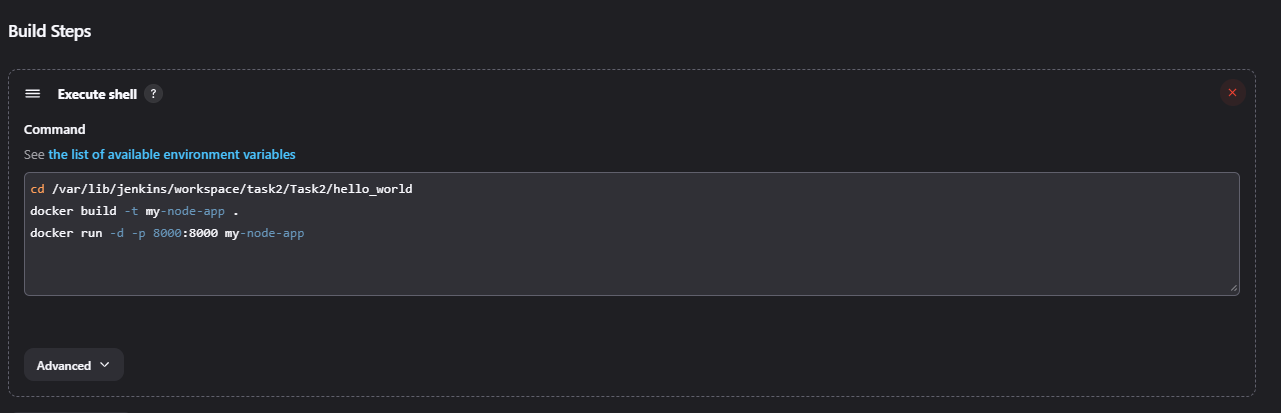


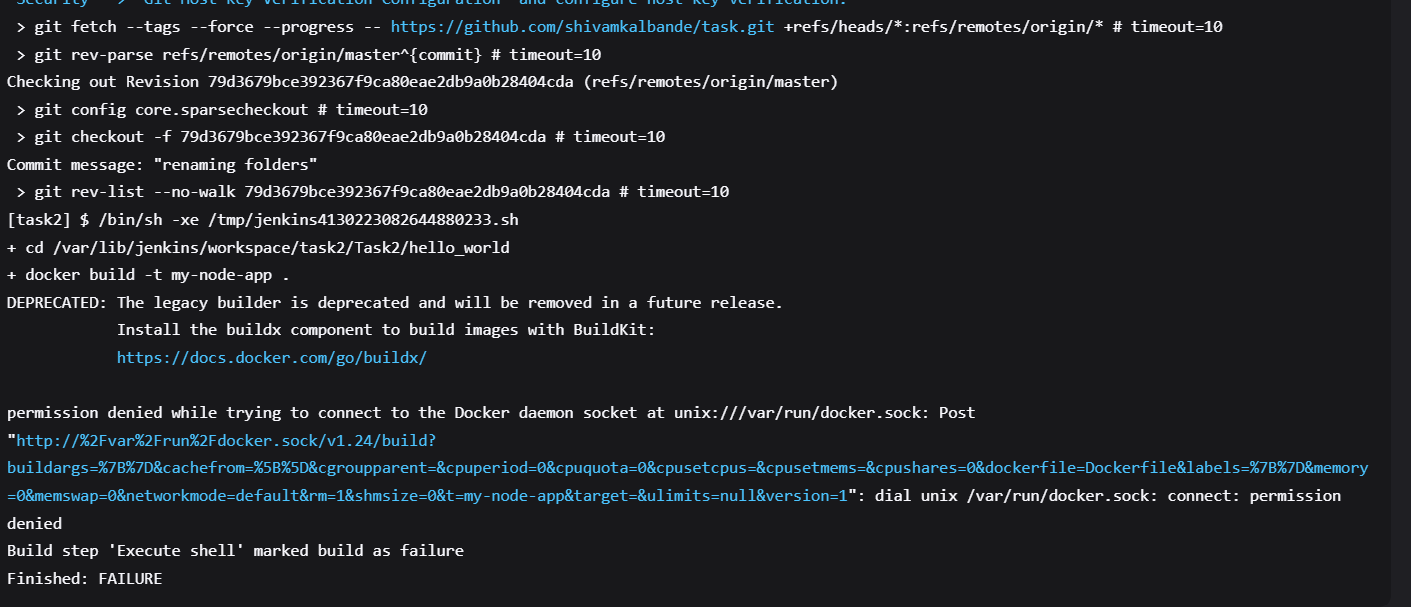
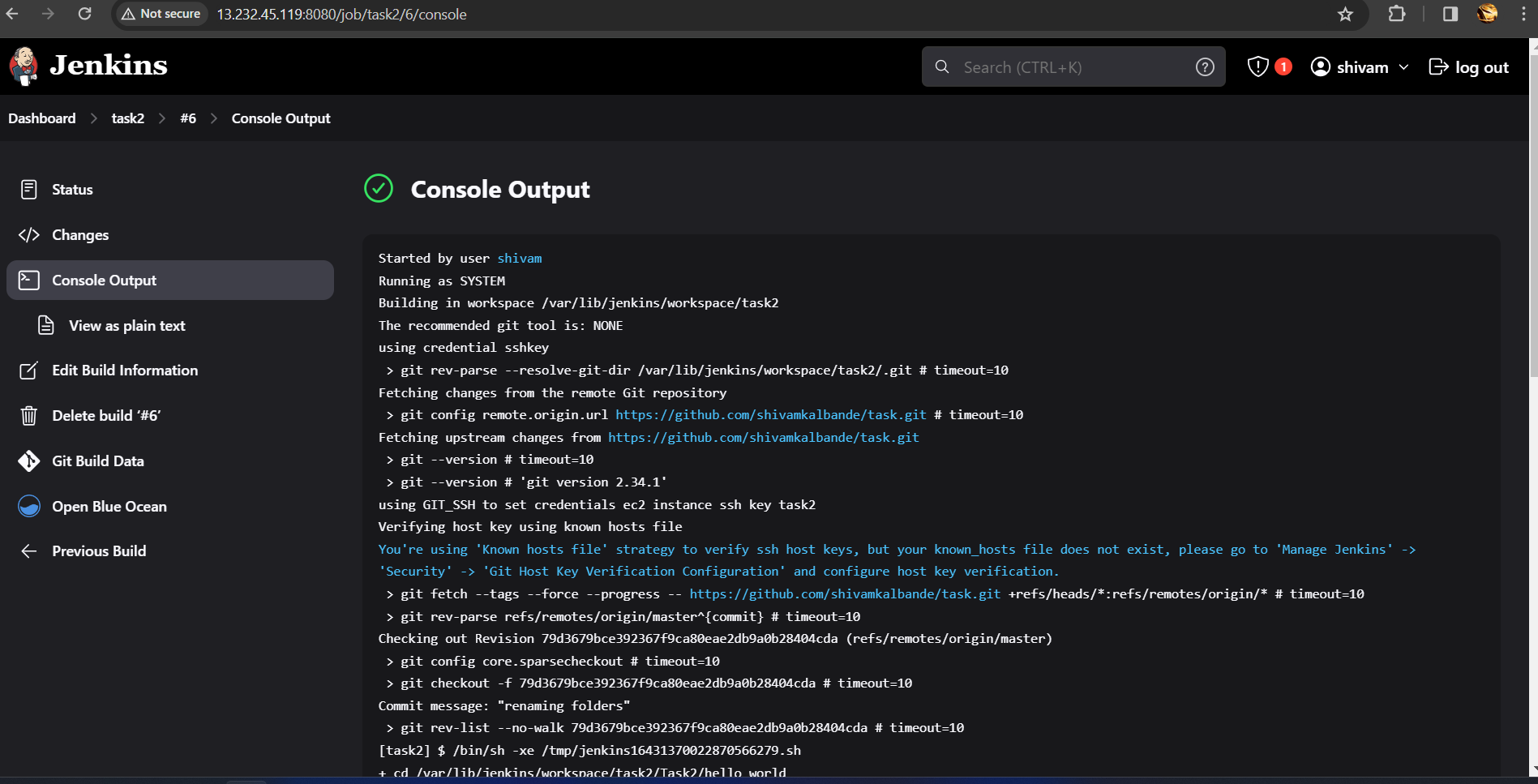












**Task 3: Minikube, Kubernetes**

• Understand Minikube

• Install Minikube either on your local laptop or on AWS EC2 instance

https://www.linuxbuzz.com/install-minikube-on-ubuntu/

$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

$ sudo install minikube-linux-amd64 /usr/local/bin/minikube

$ minikube version

$ sudo vi nginx-pod.yaml

$

$ kubectl apply -f nginx-pod.yaml

$ kubectl expose pod nginx-pod --port=80 --type=NodePort

$ sudo vi nginx-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

spec:

containers:

- name: nginx-container

image: nginx:latest

ports:

- containerPort: 80

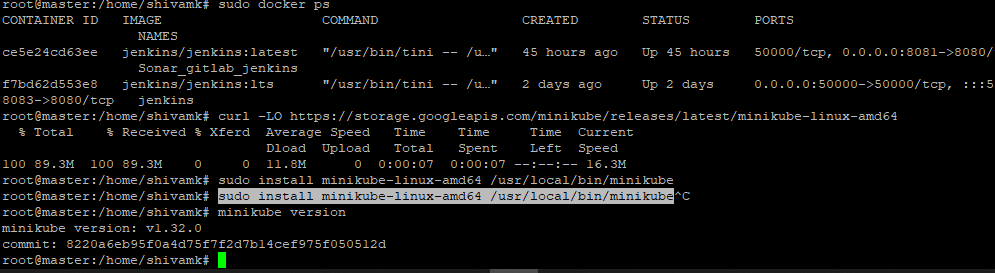
$ kubectl apply -f nginx-pod.yaml

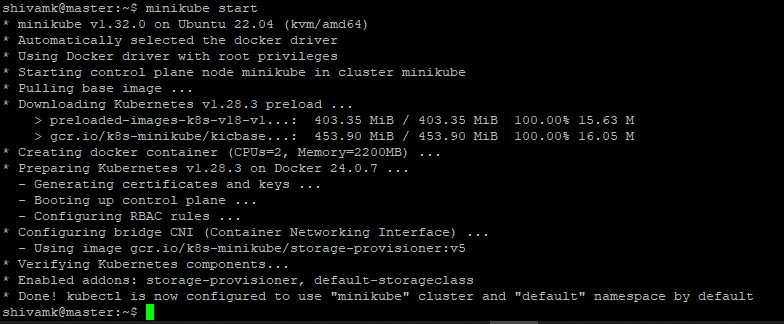
$ sudo docker ps

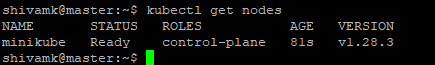
$ kubectl expose pod nginx-pod --port=80 --type=NodePort

$ kubectl get pods

$ kubectl get service







• Run Nginx named POD on minikube with container image “nginx:latest:

and expose Nginx application on port 80

sudo vi nginx-pod.yaml

kubectl apply -f nginx-pod.yaml

kubectl expose pod nginx-pod --port=80 --type=NodePort

sudo vi nginx-pod.yaml

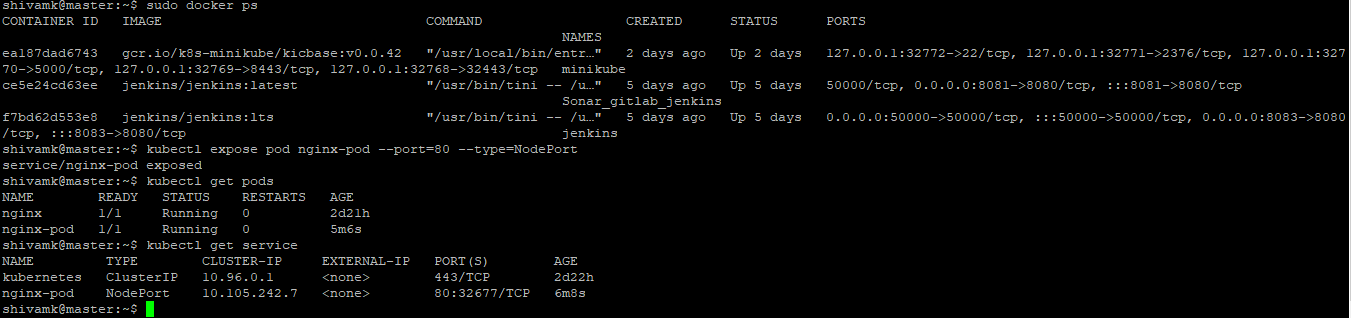
kubectl apply -f nginx-pod.yaml

sudo docker ps

kubectl expose pod nginx-pod --port=80 --type=NodePort

kubectl get pods

kubectl get service



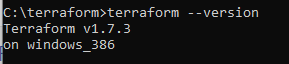
**Task 4: Terraform, AWS, Sample App**

Important Note: Do not push your AWS Access key and Secret key to GitHub

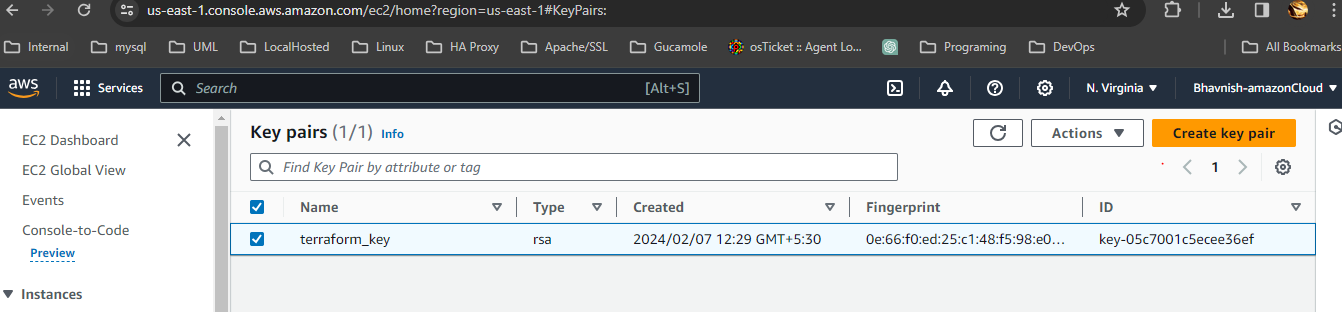
repository

• Install Terraform on your local laptop

https://releases.hashicorp.com/terraform/1.7.3/terraform\_1.7.3\_windows\_386.zip



• Create Access keys in your AWS account

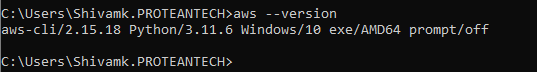


access key:xxxxxxxxxxxx

secret key:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

• Install AWS CLI on your local laptop

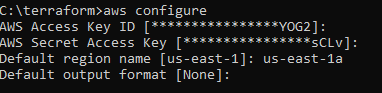
https://awscli.amazonaws.com/AWSCLIV2.msi



https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli

• Configure AWS on your local laptop using Access key and Secret key



notepad main.tf

provider "aws" {

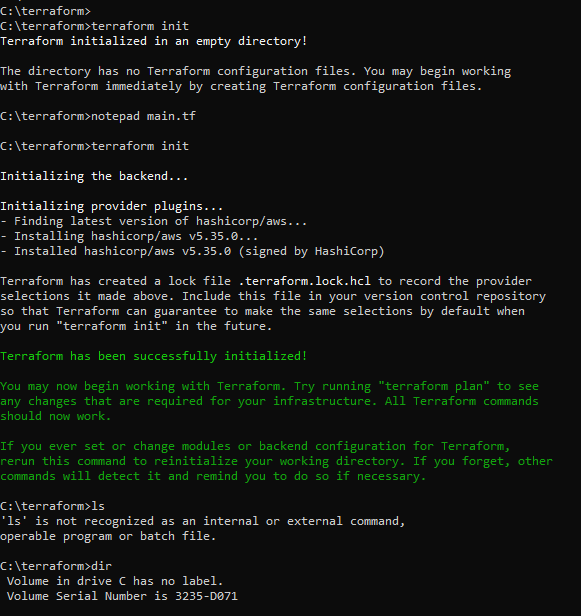
access\_key =" "

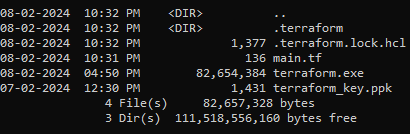
secret\_key=" "

region = "us-east-1a"

}

terraform init





• Create terraform script to create below infrastructure on AWS

* VPC

--<https://computingforgeeks.com/how-to-install-terraform-on-ubuntu/>

--[Terraform Hands-on Project — Build Your Own AWS Infrastructure with Ease using Infrastructure as Code | by Sayali Shewale | Medium](https://medium.com/@sayalishewale12/terraform-hands-on-project-build-your-own-aws-infrastructure-with-ease-using-infrastructure-as-9f17640518d7)

--[How to Build AWS VPC & Subnets using Terraform - Step by Step (spacelift.io)](https://spacelift.io/blog/terraform-aws-vpc)

--https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/vpc

provider "aws" {

access\_key =" "

secret\_key=" "

region = "us-east-1"

}

##Retrive the AZ where we want to create network resources

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

#VPC Resources

resource "aws\_vpc" "main" {

cidr\_block = "10.11.0.0/16"

enable\_dns\_support = true

enable\_dns\_hostnames = true

tags = {

Name = "Test-VPC"

}

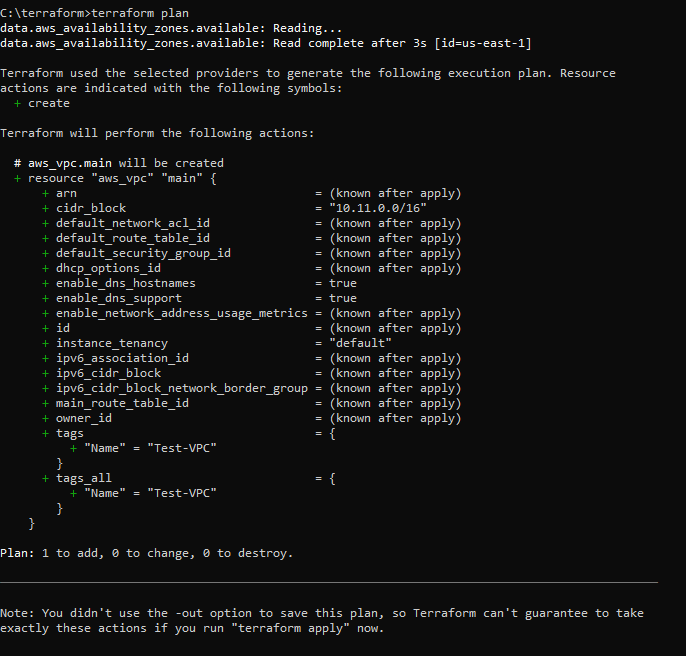
}

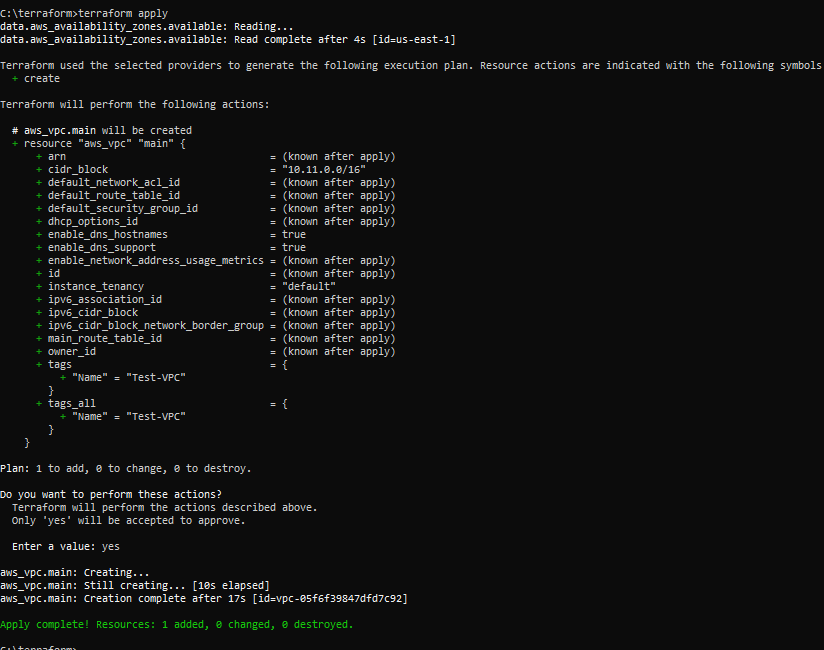
>> terraform plan

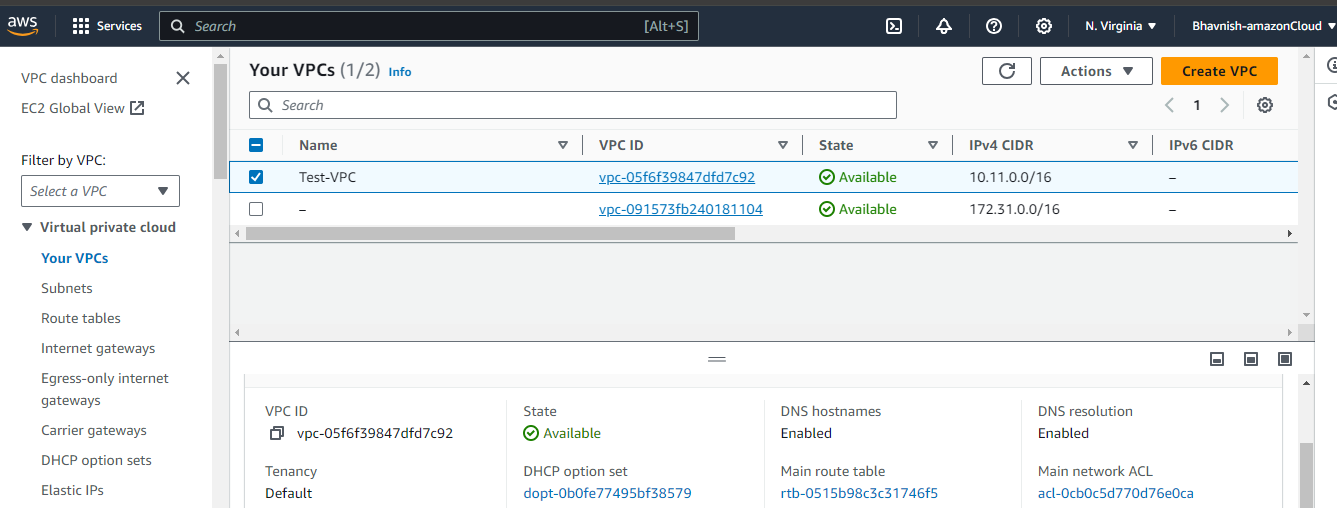
>> terraform validate

>> terraform apply

--YES to apply







* Internet Gateway

provider "aws" {

access\_key =" "

secret\_key=" "

region = "us-east-1"

}

##Retrive the AZ where we want to create network resources

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

#VPC Resources

resource "aws\_vpc" "main" {

cidr\_block = "10.11.0.0/16"

enable\_dns\_support = true

enable\_dns\_hostnames = true

tags = {

Name = "Test-VPC"

}

}

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

vpc\_id = aws\_vpc.main.id

cidr\_block = "10.11.1.0/24"

tags = {

Name = "Public Subnet"

}

}

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

vpc\_id = aws\_vpc.main.id

cidr\_block = "10.11.2.0/24"

map\_public\_ip\_on\_launch = false

tags = {

Name = "Private Subnet"

}

}

resource "aws\_internet\_gateway" "gw" {

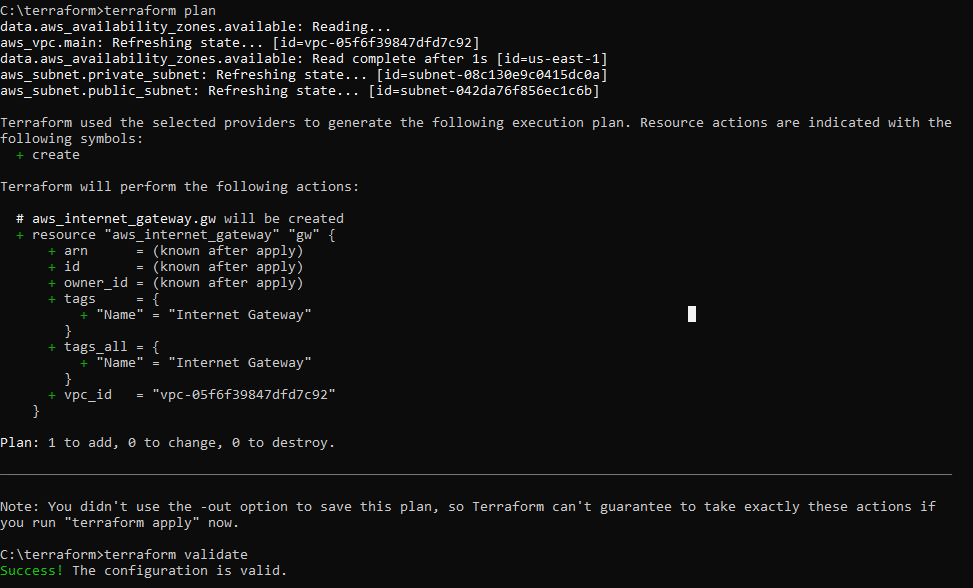
vpc\_id = aws\_vpc.main.id

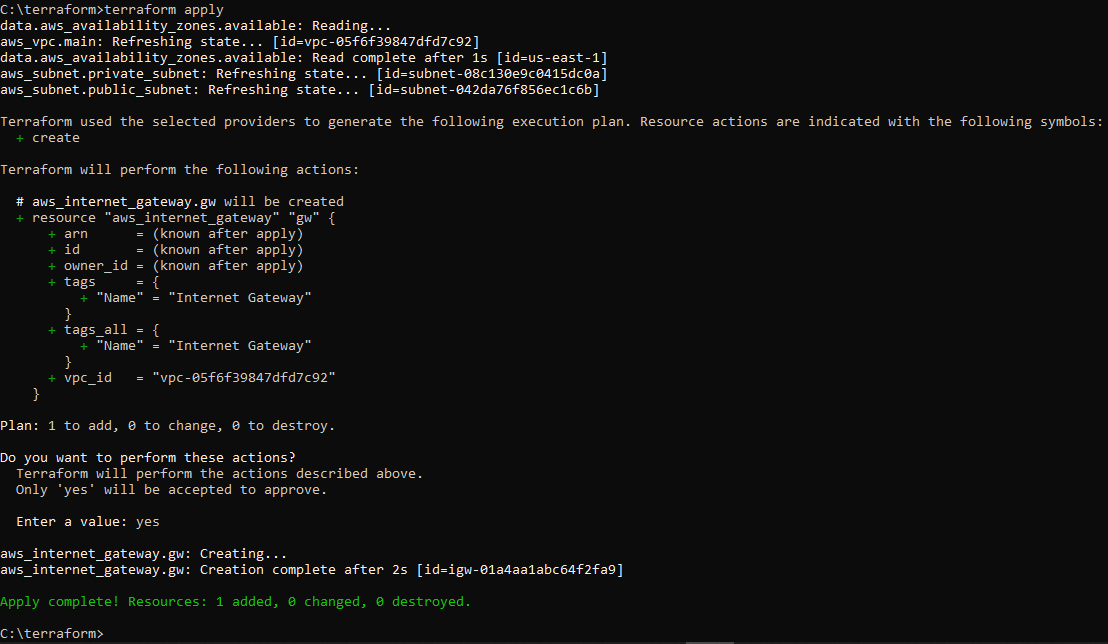
tags = {

Name = "Internet Gateway"

}

}





* Public Subnet

provider "aws" {

access\_key =" "

secret\_key=" "

region = "us-east-1"

}

##Retrive the AZ where we want to create network resources

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

#VPC Resources

resource "aws\_vpc" "main" {

cidr\_block = "10.11.0.0/16"

enable\_dns\_support = true

enable\_dns\_hostnames = true

tags = {

Name = "Test-VPC"

}

}

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

vpc\_id = aws\_vpc.main.id

cidr\_block = "10.11.1.0/24"

tags = {

Name = "Public Subnet"

}

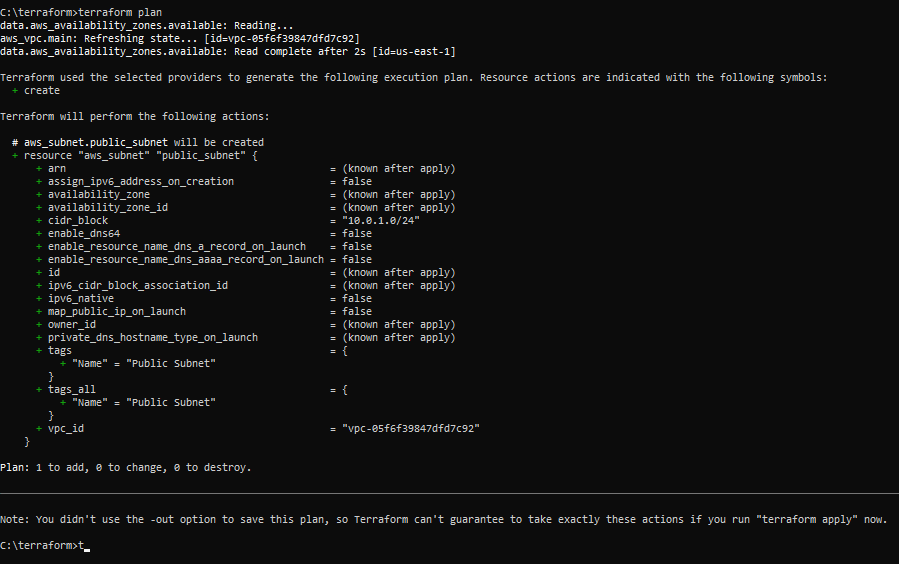
}

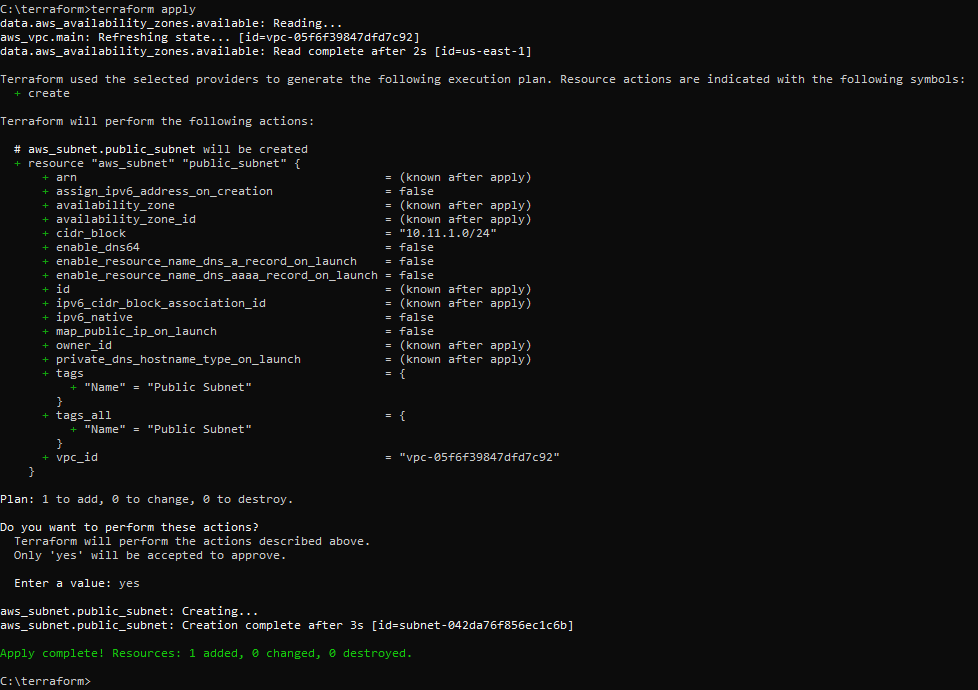
>> terraform plan

>> terraform validate

>> terraform apply

--YES to apply





* Private Subnet

provider "aws" {

access\_key =" "

secret\_key=" "

region = "us-east-1"

}

##Retrive the AZ where we want to create network resources

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

#VPC Resources

resource "aws\_vpc" "main" {

cidr\_block = "10.11.0.0/16"

enable\_dns\_support = true

enable\_dns\_hostnames = true

tags = {

Name = "Test-VPC"

}

}

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

vpc\_id = aws\_vpc.main.id

cidr\_block = "10.11.1.0/24"

tags = {

Name = "Public Subnet"

}

}

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

vpc\_id = aws\_vpc.main.id

cidr\_block = "10.11.2.0/24"

tags = {

Name = "Private Subnet"

}

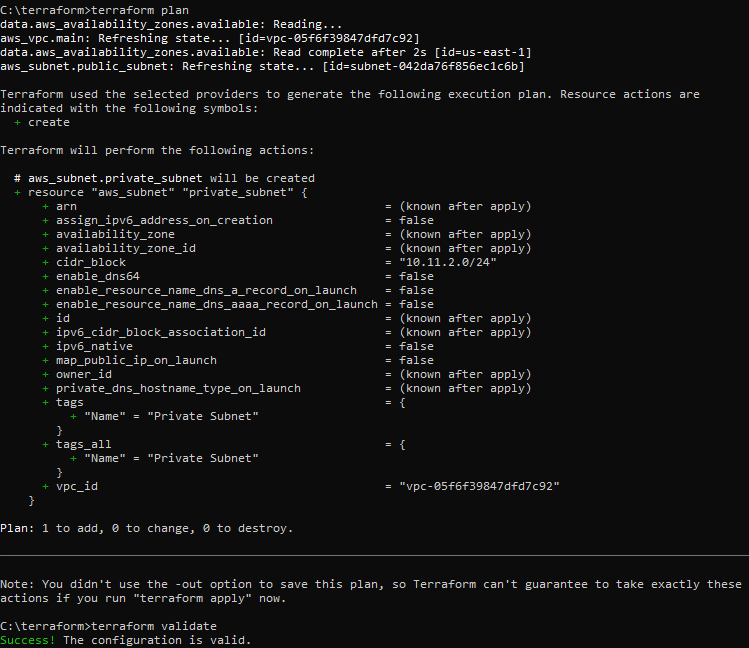
}

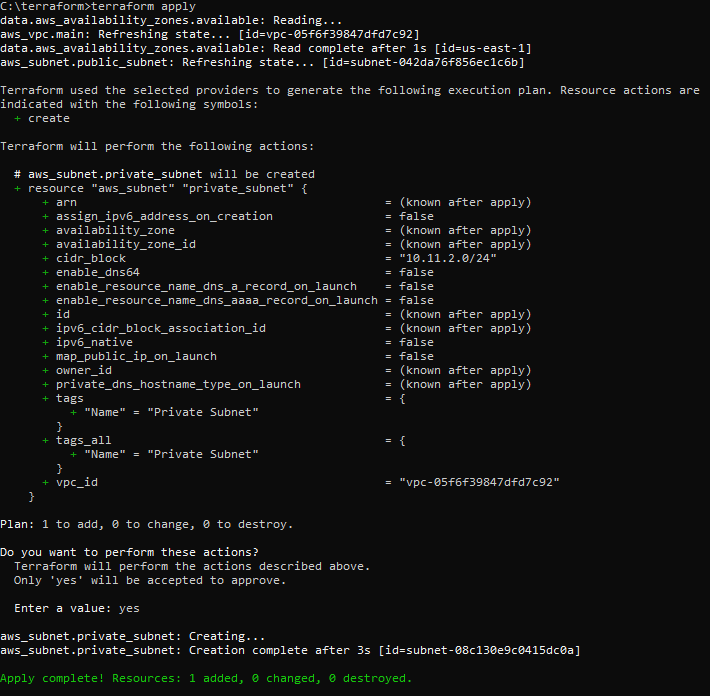
>> terraform plan

>> terraform validate

>> terraform apply

--YES to apply





* Route Table

provider "aws" {

access\_key =" "

secret\_key=" "

region = "us-east-1"

}

##Retrive the AZ where we want to create network resources

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

#VPC Resources

resource "aws\_vpc" "main" {

cidr\_block = "10.11.0.0/16"

enable\_dns\_support = true

enable\_dns\_hostnames = true

tags = {

Name = "Test-VPC"

}

}

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

vpc\_id = aws\_vpc.main.id

cidr\_block = "10.11.1.0/24"

tags = {

Name = "Public Subnet"

}

}

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

vpc\_id = aws\_vpc.main.id

cidr\_block = "10.11.2.0/24"

map\_public\_ip\_on\_launch = false

tags = {

Name = "Private Subnet"

}

}

resource "aws\_internet\_gateway" "gw" {

vpc\_id = aws\_vpc.main.id

tags = {

Name = "Internet Gateway"

}

}

resource "aws\_route\_table" "main" {

vpc\_id = aws\_vpc.main.id

route {

cidr\_block = "0.0.0.0/0"

gateway\_id = aws\_internet\_gateway.gw.id

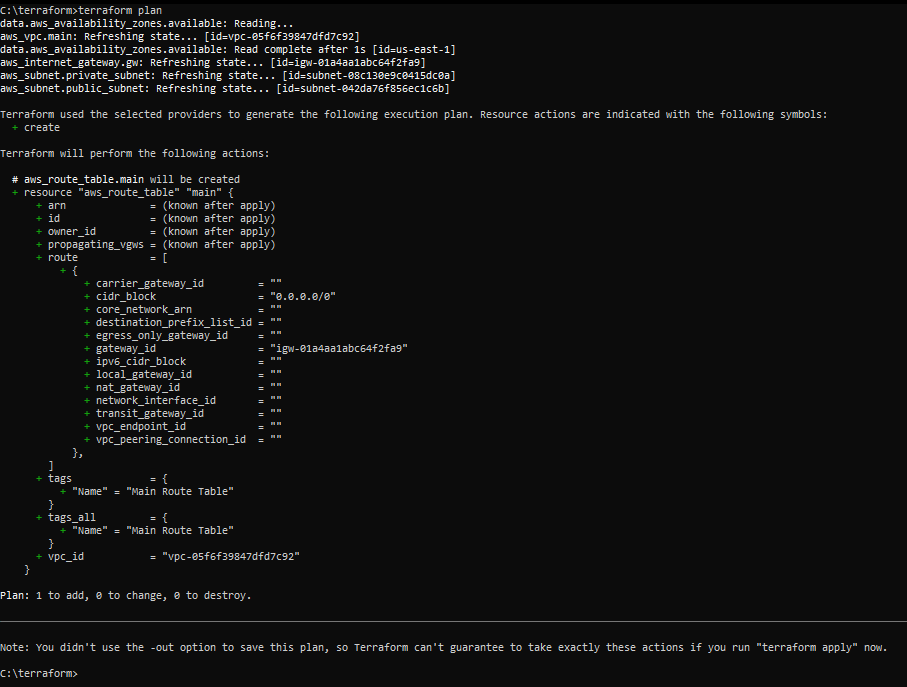
}

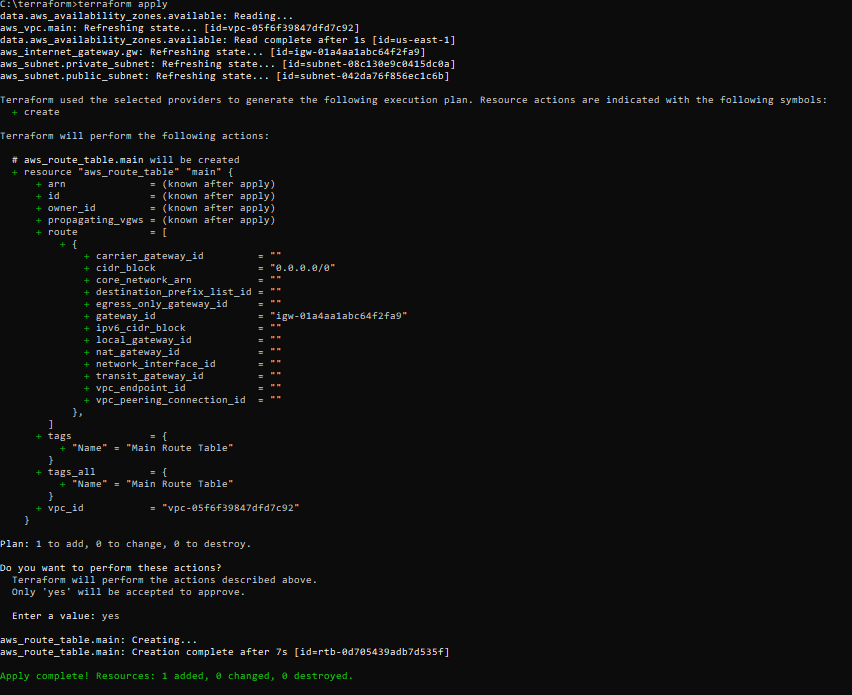
tags = {

Name = "Main Route Table"

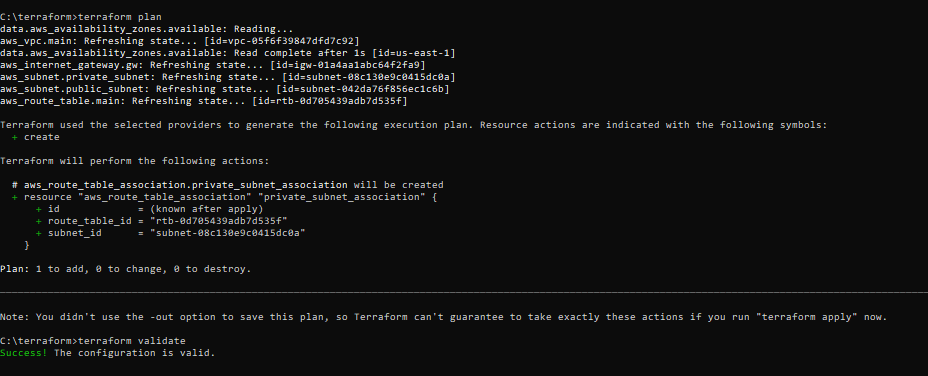
}

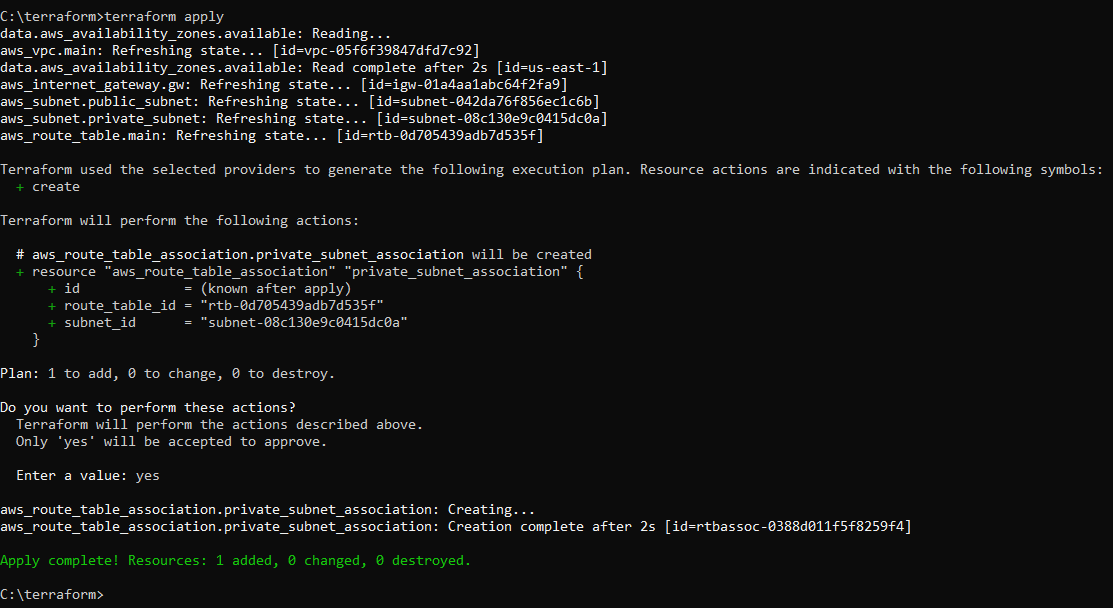
}

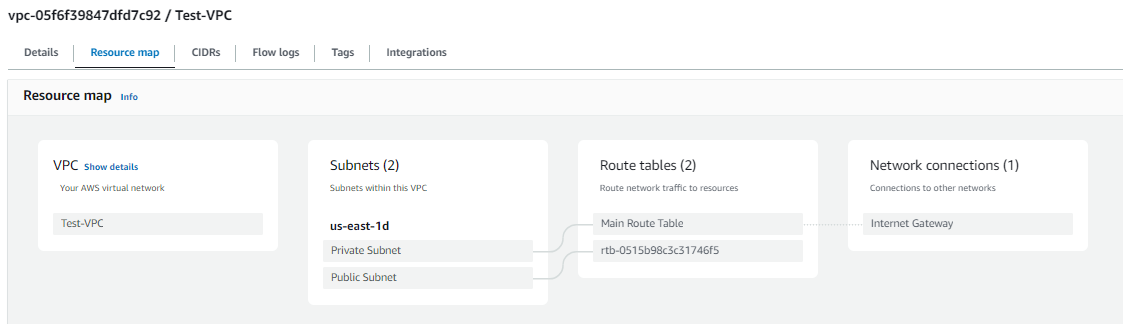




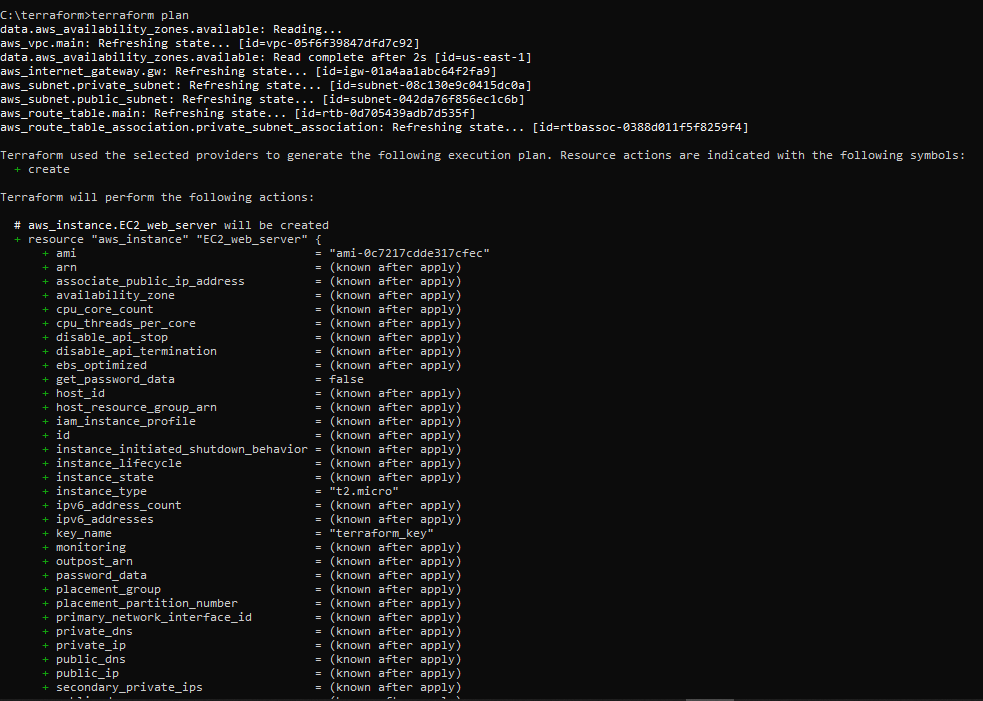
* Route Table Association with Subnets

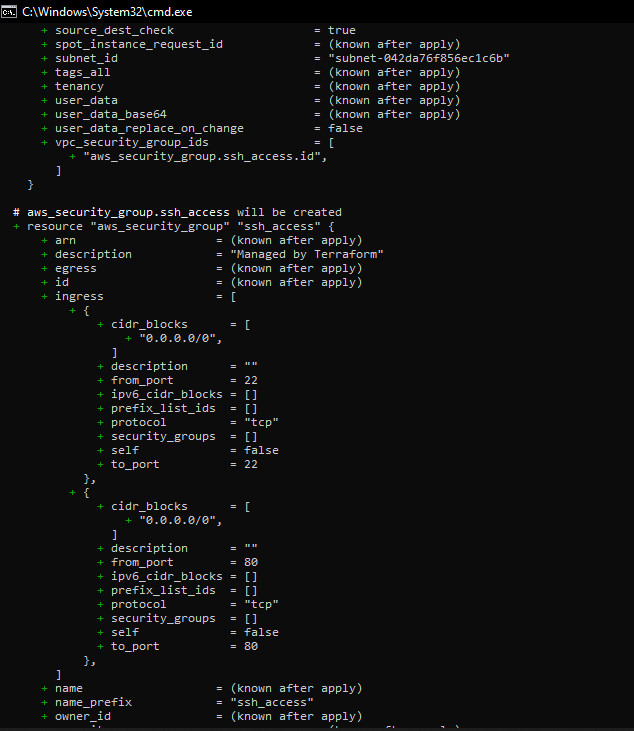


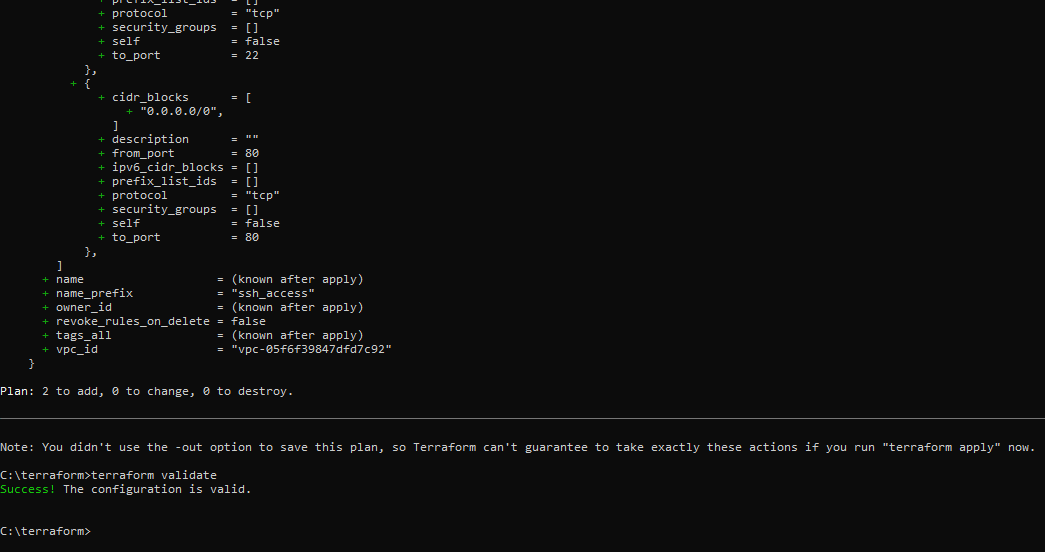


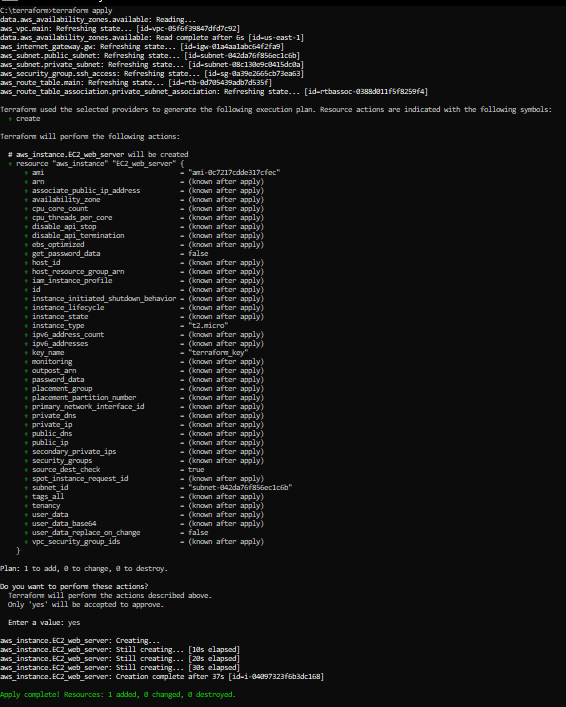


* EC2 Instance
* Security Group









* Elastic IP

resource "aws\_eip" "example" {

instance = aws\_instance.EC2\_web\_server.id

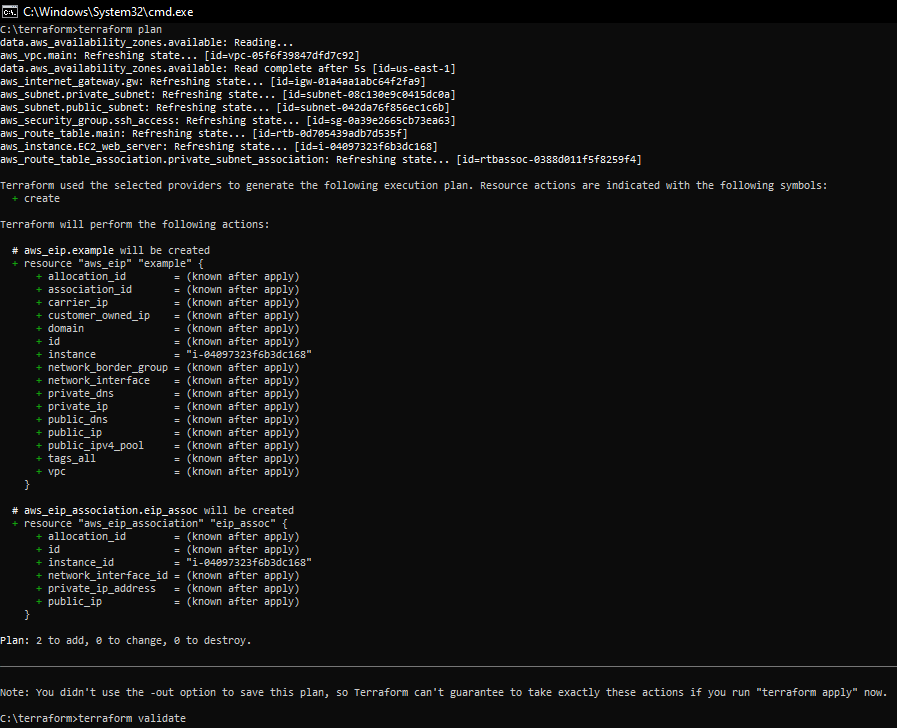
}

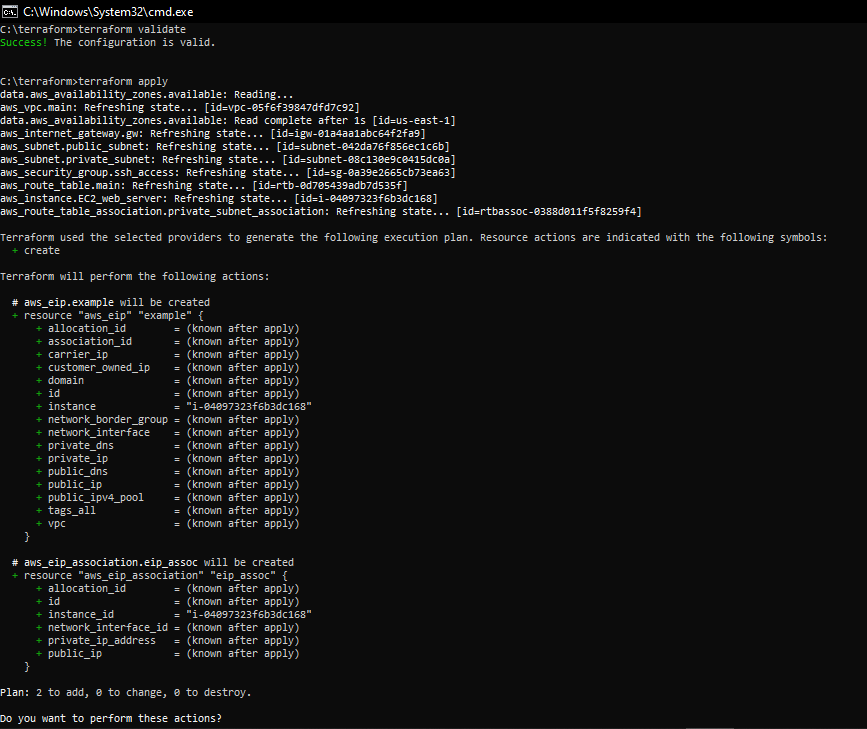
resource "aws\_eip\_association" "eip\_assoc" {

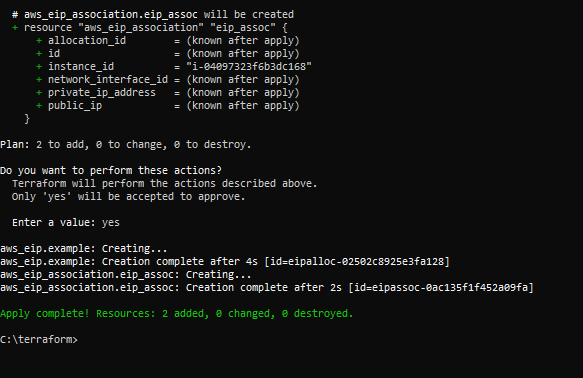
instance\_id = aws\_instance.EC2\_web\_server.id

allocation\_id = aws\_eip.example.id

}







• You need to pass User-data script to EC2 Instance resource group

Section in terraform to install and run sample application using terraform

• Refer this blog for this task, also read my comment below this blog.

<https://medium.com/@sayalishewale12/terraform-hands-on-projectbuild-your-own-aws-infrastructure-with-ease-using-infrastructure-as9f17640518>

DESTROYING the Resources

------------------------------------------

